Information in this document is subject to change without notice and does not represent a commitment on the part of Soft Warehouse, Inc. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium, for any purpose other than the purchaser's personal use.

Second printing, January 1988
Copyright © 1979, 1980, 1983, 1985, 1986, 1987 by Soft Warehouse, Inc.
All Rights Reserved Worldwide

No part of this document may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the express written permission of Soft Warehouse, Inc.

If you have comments about or difficulties with this documentation or the software, please complete the muLISP-87 Problem Report at the back of this manual and return it to Soft Warehouse, Inc., 3615 Harding Avenue, Suite 505, Honolulu, Hawaii, 96816, U.S.A.

muLISP is a trademark of Soft Warehouse, Inc.

Microsoft, the Microsoft logo, MS, and MS-DOS are registered trademarks of Microsoft Corporation.

ADM-3A is a trademark of Lear Siegler Incorporated.

Hewlett-Packard is a registered trademark of Hewlett-Packard Company.

IBM is a registered trademark of International Business Machines Corporation.

WordStar is a registered trademark of MicroPro International Corporation.

Zenith is a registered trademark of Zenith Radio Corporation.

PREFACE

The LISP computer programming language is based on a paper by John McCarthy entitled Recursive Functions of Symbolic Expressions and Their Computation by Machine [1960]. Since its first implementation at the Massachusetts Institute of Technology in the early 1960's, LISP has remained the "machine language" of the Artificial Intelligence (AI) community.

Going beyond its AI roots, LISP has evolved into a powerful general-purpose computer language. It is *the language of choice* for implementing state-of-the-art software because:

LISP is an *applicative* language that encourages a functional approach to designing large software systems.

LISP is a *recursive* language that makes it an ideal formalism for describing complex mathematical concepts.

LISP is a *list processing* language that can effectively simulate a wide range of real-world problems on a computer by making transformations on the lists and binary trees that model the problems.

LISP is an *extensible* language to which you can add your own functions and control constructs.

LISP uses *dynamic memory management* to automatically recycle and reallocate memory resources according to the changing demands of programs as they run.

LISP is a *data-driven* language that allows data structures to be created and then executed as a program.

LISP is an *incremental* language that supports top-down, bottom-up, *and* middle-out programming styles as appropriate.

LISP is an *interactive* language that provides the software developer with a powerful programming environment essential for intelligent human-machine communication.

For these reasons LISP is ideal for projects in such diverse areas as natural language translation, robotics, game playing, expert systems, theorem proving, compiler writing, systems programming, number theory, medical diagnosis, and program verification.

muLISP-87 is the culmination of over ten years of effort into designing a high performance, general purpose LISP system for personal computers. While incorporating many of the ideas of modern mainframe systems, especially Common LISP (see Steele, [1984]), muLISP remains a *lean and mean* LISP system. The design philosophy behind muLISP has always been to provide an efficient and compact kernel of primitively defined functions and special forms. This leaves sufficient memory for building powerful utility libraries, programming environments, and application programs on top of this machine language kernel.

muLISP was conceived and implemented by Albert D. Rich of Soft Warehouse, Inc. The Irrational Function package was written by David R. Stoutemyer of Soft Warehouse, Inc. The Flavors and Dynamic Closure packages were written by Peter Ohler. The Eights Puzzle was written by Peter A. Harada.

The design of muLISP was influenced by the suggestions of Richard Zipple, David R. Stoutemyer, Samuel E. Rhoads, Peter van Lith, Peter A. Harada, Martin Griss, Ralph S. Freese, Peter Deutsch, and many others. The clarity and completeness of this manual are due in large part to the patience and persistence of Joan H. Rich and David R. Stoutemyer.

Albert D. Rich
Applied Logician
Soft Warehouse, Inc.
May 1987

TABLE OF CONTENTS

Title Page i
Copyright Notice ii
Preface iii
Table of Contents v

1. INTRODUCTION

1.1 Major Features 1-3
1.2 muLISP System Contents 1-10
1.3 System Requirements 1-11
1.4 Master Diskette Backup 1-12
1.5 The IBM PC Keyboard 1-14

2. muLISP OPERATING PROCEDURES

2.1 System Load and Run 2-1
2.2 The Read-eval-print Loop 2-2
2.3 Line Editing Procedures 2-4
2.4 Programming Fundamentals 2-6
2.5 Reading Source Files 2-9
2.6 Environment SYS Files 2-10

3. LEARNING muLISP

3.1 Demonstration Programs 3-1
3.2 muLISP Tutorial System 3-4

4. THE muLISP INTERPRETER

4.1 Data Structures 4-1
4.2 Memory Management 4-6
4.3 The Break Package 4-9
4.4 Console Interrupts 4-11
4.5 Error Messages 4-12

5. THE muLISP PRIMITIVES

5.1 Selector and Searcher Functions 5-3
5.2 Constructor Functions 5-18
5.3 Modifier Functions 5-38
5.4 Recognizer Functions 5-55
5.5 Comparator Functions 5-61
5.6 Logical Functions 5-66
5.7 Assignment Functions 5-69
5.8 Property Functions 5-76
5.9 Flag Functions 5-80
5.10 Definition Primitives 5-82
5.11 Print Name Functions 5-90
5.12 Numerical Functions 5-110
5.13 Input Primitives 5-159
5.14 Output Primitives 5-177
5.15 File System Primitives 5-189
5.16 Evaluation Functions 5-212
5.17 Control Constructs 5-232
5.18 Memory Management Functions 5-250
5.19 User Interface Functions 5-256
5.20 Hardware Interface Functions 5-266
5.21 Console Screen Primitives 5-276
5.22 Miscellaneous Functions 5-288

6. THE muLISP LIBRARY

6.1 The Array Package 6-1
6.2 The Structure Package 6-6
6.3 The Flavors Package 6-10
6.4 The Turtle Graphics Package 6-18
6.5 The Mouse Package 6-23
6.6 The Closure Package 6-29

7. THE PROGRAM DEVELOPMENT SYSTEM

7.1 The muLISP Editor 7-2
7.2 The muLISP Debugger 7-20
7.3 The muLISP Window Manager 7-26

8. THE muLISP COMPILER

8.1 Compiler Functions 8-3
8.2 Basic Compilation Procedure 8-7
8.3 Compiled Function Performance 8-11
8.4 Errors in Compiling Functions 8-15
8.5 Potential Problem Areas 8-17

9. MACHINE LANGUAGE ROUTINES

9.1 Data Representation 9-2
9.2 Argument Passing 9-5
9.3 Calling Primitive Routines 9-7
9.4 Loading and Linking 9-9

10. LISP AND AI REFERENCES

10.1 Professional Societies 10-1
10.2 Periodicals 10-2
10.3 Books 10-3

11. GLOSSARY

The muLISP Compiler is an optional package that can be purchased from Soft Warehouse, Inc.

APPENDIXES

Appendix A: IBM PC Extended Function Keys A-1
Appendix B: muLISP Base Page A-2
Appendix C: Global Storage Area A-11
Appendix D: Sample Assembly Language Routine A-13
Appendix E: Wordstar-like Editor Commands A-14
Appendix F: Emacs-like Editor Commands A-16
Appendix G: muLISP-86 to '87 Source File Conversion A-18

INDEXES

Concept Index I-1
Function Index I-9

muLISP-87 PROBLEM REPORT

1. INTRODUCTION

Congratulations on your purchase of muLISP-87. It is a powerful LISP programming environment for artificial intelligence (AI) research efforts on personal computers.

As might be expected, some degree of study and patience is required to make maximum use of muLISP as an AI development tool. The best way is through the *hands-on* experience provided by the muLISP Tutorial System (see Chapter 3). It is an on-line interactive system that teaches the *fundamentals* of LISP programming. Chapter 10 lists many good LISP books and periodicals that cover more *advanced* AI programming techniques.

This manual is meant to serve as a *reference guide* for muLISP. Every effort has been made to make the information presented in this manual as clear, comprehensive, and accessible as possible. However, it is *not* a tutorial on muLISP or the LISP programming language.

The information presented in this manual is arranged as follows:

Chapter 1. Introduction summarizes the major features of muLISP, the files on the muLISP master diskette, the computer requirements necessary to run muLISP, the backup procedure for the master muLISP diskette, and the notation used on the IBM PC keyboard.

Chapter 2. Operating Procedures describes how to load and run muLISP, how to enter and edit commands, how to define new functions, how to load source files, and how to load and save SYS files.

Chapter 3. Learning muLISP describes how to load and play the demonstration game programs and how to start the muLISP Tutorial System.

Chapter 4. The muLISP Interpreter describes the primitive muLISP data structures, the memory management system, the break package, the console interrupts, and the error messages.

Chapter 5. The muLISP Primitives describes the functions, special forms, macros, and control variables defined in the muLISP machine language kernel and in the COMMON.LSP and IRRATNAL.LSP source files.

Chapter 6. The muLISP Library describes the functions, macros, and control variables defined in the source files making up the library of packages. Packages are provided for implementing arrays, structures, flavors, turtle graphics, mouse interfaces, and dynamic closures.

Chapter 7. The muLISP Program Development System describes the window manager, the editor, the debugger, and how to write your own windows applications.

Chapter 8. The muLISP Compiler describes how to load and make the best use of muLISP's optional native-code compiler.

Chapter 9. Machine Language Routines describes how to write, load, and link machine language routines to muLISP.

Chapter 10. LISP and AI References lists societies, periodicals, and books relevant to LISP and AI.

Chapter 11. Glossary defines words relevant to personal computers and AI that are *not* defined elsewhere in this manual.

1.1 MAJOR FEATURES

muLISP consists of a machine language kernel, a window based program development system, a library of packages, several demonstration game programs, a tutorial system, and documentation.

1.1.1 The Machine Language Kernel

The machine language kernel is distributed as the executable COM file MULISP.COM. It includes the muLISP interpreter, garbage collector, and over 400 functions, special forms, and control variables. They are defined in machine language for maximum efficiency.

Since the kernel consumes less than 50 Kilobytes of memory, it leaves a large amount of computer memory free for application programs. Although muLISP can run in as little as 256 Kilobytes of memory, it can take advantage of up to 640 Kilobytes.

Extremely fast program execution speeds have been achieved through the use of such techniques as shallow variable binding, address typed data structures, and a closed pointer universe. The following summarizes the capabilities provided by the machine language kernel:

List Processing List processing is the foundation on which all LISP programs are built. The kernel provides a vast array of functions for accessing, constructing, recognizing, comparing, counting, searching, substituting, and modifying the elements of lists.

Merging and Sorting The highly efficient functions for merging and sorting lists accept user supplied predicates. The fast, stable list-merge sort requires time proportional to $n \log n$, where n is the length of the list being sorted.

Arithmetic Operations In addition to "infinite" precision integer arithmetic, muLISP implements both exact *and* approximate rational (floating-point) arithmetic. Both the precision used for arithmetic operations and the radix used to express numbers is adjustable.

Function Mapping The mapping functions and mapping predicates provide an elegant means of iteratively testing and/or manipulating the elements of any number of lists.

Character and String Processing The wide selection of character and string processing functions including substring extraction, string searches, case conversion, and comparisons (both case sensitive and insensitive). A hashing technique is used to rapidly intern new print name strings.

Input/Output Facilities The I/O functions can read and write character data in the form of LISP expressions, strings, lines, characters, and bytes to and from the console, the printer, and open files. Control variables determine the use of case conversion, escape characters, and radix base.

Macro Characters The syntax recognized by the LISP reader can be modified by defining macro characters (terminating, non-terminating, and comment). For instance, the backquote character greatly simplifies macro definitions.

File System Both sequential and random access to any number of open files is fully supported. File system functions open and close files, rename and delete files, change default drives and paths, determine the date of files, and provide directory information.

Environment Capture The entire muLISP environment can be captured as a compact memory image file that can be rapidly loaded at a later time.

Control Constructs The kernel provides a full range of program control constructs. The primitively defined control constructs include an "implicit PROGN", COND, IF, PROGN, PROG1, LOOP, and RETURN. The macro defined control constructs include PROG2, WHEN, UNLESS, CASE, LET, LET*, DO, DO*, DOLIST, and DOTIMES. Finally, the "implicit-COND" muLISP uses when evaluating function bodies can greatly improve a program's performance, code density, and readability.

Non-local Exits The special forms CATCH, THROW, and UNWIND-PROTECT provide a structured, non-local exit mechanism that

greatly simplifies the handling of errors and exceptional conditions in application programs.

Macros Macros can be expanded at run time or pseudo-compile time to produce code that effectively supplements the already rich collection of primitively defined control constructs. For example, the macros SETF, INCF, and DECF allow a consistent syntax to be used when updating generalized variables. DEFMACRO destructures Lambda lists in accordance with Common LISP.

Argument Passing Normally, a function accepts only a fixed number of arguments and the arguments are evaluated before being passed to the function. In muLISP, a function can also be defined to accept any number of arguments and/or to *not* evaluate its arguments.

Pseudo-code Compiler When a function is defined, its definition is automatically compiled into a very compact pseudo-code called D-code. Since the inverse process of decompiling is also automatic, the whole process is invisible to the user. Compilation not only improves the speed of the interpreter but requires only 1/3 as much memory as storing definitions as linked lists.

Machine Language Routines If necessary for some special application or performance need, additional machine language routines can be added to the kernel. The necessary calling conventions and the loading and linking procedure are described elsewhere in this manual.

Memory Management A two-pass, compacting garbage collector automatically reclaims unreferenced data structures, thus freeing for reuse the space these structures consume. When necessary, a memory reallocator also dynamically adjusts boundaries to provide more room for heavily used data spaces. In short, the muLISP memory management system automatically *maximizes* available memory resources for application programs while making it almost *invisible* to the programmer.

Subprocess Invocation muLISP can be temporarily suspended in order to load and run any MS-DOS program including the MS-DOS command processor. Data can be exchanged with the program

through files and/or exit codes. When the subprocess terminates, muLISP resumes execution.

Generic MS-DOS muLISP is designed to run on any computer that operates under PC-DOS[tm] *or* MS-DOS[tm]. This includes the IBM PC and PS/2 series computers, and the MS-DOS computers made by AT&T, Compaq, Hewlett-Packard, Kaypro, Leading Edge, Morrow, Texas Instruments, NEC, Tandy, Toshiba, Zenith, and many others.

IBM PC Support If running on an IBM PC or compatible computer, muLISP automatically takes advantage of the PC's hardware including blinking characters, high and low intensity, inverse video, partial-screen scrolling, color, cursor shape, tone generation, and CGA and EGA color graphics.

1.1.2 The Program Development System

muLISP provides the user with a powerful window-based programming environment that facilitates the interactive development of sophisticated AI software.

Window Manager The muLISP window manager provides a flexible, menu-driven interface for window-oriented application programs. It includes facilities for opening, splitting, closing, and changing windows. The muLISP editor and debugger are fully integrated into the window manager and serve as good examples of how to write window-oriented application programs.

Source File Editor The muLISP editor is a powerful screen-oriented text editor that has been enhanced to facilitate the editing of LISP source files by incorporating such features as expression evaluation, blinking parentheses, and undelete. With just a single key stroke, you can suspend the editing of a source file and switch to a debug window or even another edit window.

Program Debugger The muLISP debugger displays function trace output continuously or in single-step mode. In single-step mode it is easy to examine and/or modify intermediate results. Screen-oriented programs can be debugged more easily since trace output is sent to a debug window separate from the program's output window.

Runtime System After loading or compiling a muLISP application program, the program together with the muLISP Runtime System can be saved as a directly executable MS-DOS COM or EXE file. This file can be made commercially available if you agree to the terms of the muLISP Runtime License Agreement *and* pay a modest royalty for the use of the Runtime System. The royalty rate is based on a sliding scale determined by the number of systems purchased. For details, write Soft Warehouse, Inc., 3615 Harding Avenue, Suite 505, Honolulu, Hawaii, 96816, U.S.A.

Native Code Compiler The muLISP compiler is an *optional* package that can be purchased from Soft Warehouse, Inc. This fully *incremental* compiler transforms function definitions directly into native machine code. It can improve the performance of speed-critical functions in an application program by a factor of 2 or more.

1.1.3 The Package Library

muLISP includes a library of optional packages that complement the facilities provided by the machine language kernel.

Common LISP Package Common LISP is rapidly becoming the standard for the LISP programming language on both mainframe and microcomputers. When this and the other packages are loaded, muLISP provides the user with over 450 Common LISP functions, special forms, macros, and control variables.

Irrational Function Package This package provides functions for approximating the exponential, logarithmic, trigonometric, and inverse trigonometric functions to any desired degree of accuracy.

Array Package This package provides a set of functions and macros for creating, accessing, and recognizing multi-dimensional arrays.

Structure Package This package provides the DEFSTRUCT macro for creating named and unnamed record structures. Defining a structure in effect creates a new data type complete with accessors, constructors, recognizers, and copiers.

Flavors Package This package provides a set of functions and macros that facilitate object-oriented programming in muLISP. Patterned after the flavors package found on LISP machines, it supports multiple inherited methods and instance variables, before and after demons, and whopper methods.

Graphics Package This package provides functions for doing point, line, and turtle graphics. Included as examples are functions for recursively drawing the popular "C" and "Dragon" curves.

Mouse Package This package provides functions for reading the position and button status of the Microsoft Mouse™.

Closure Package This package provides functions for creating and manipulating dynamic closures of functions over a list of variables whose values are local to the closures.

1.1.4 The Demonstration Game Programs

muLISP includes several interesting game and puzzle programs that demonstrate good muLISP and AI programming techniques.

Animal This word game builds and uses a discrimination net to logically determine what animal the player is thinking of by asking a series of yes-or-no questions.

Doctor This game, sometimes called Eliza, turns your computer into a psychologist of the Rogerian school of psychology, in which the doctor lets the patients resolve their own problems by asking leading questions.

Eights This program uses heuristic search techniques to "slide" eight "tiles" that are randomly arranged in a 3 by 3 "tray" into a standard configuration.

Tower of Hanoi This program uses character graphics and sound effects to display the recursive solution to this famous puzzle.

Metamind This program can play either side of this code breaking game. The object is to determine the colors and arrangement of four objects based on clues provided by the code maker.

1.1.5 The Tutorial System and Documentation

Tutorial System The interactive muLISP Tutorial System is a friendly, on-line introduction to LISP programming. Topics include the basic LISP data structures and primitives, recursion and iteration, numerical programming techniques, and turtle graphics.

Help Facility The muLISP on-line help facility provides ready information on the arguments and/or initial value of the muLISP primitives and where in the manual to find more detailed information.

Reference Manual The 480 page muLISP-87 Reference Manual provides detailed explanations and numerous examples of the functions, macros, special forms, and control variables defined in the muLISP kernel and in the library of packages.

Soft Warehouse Newsletter Published by the authors of muLISP, the Newsletter provides users with the latest information on muLISP and also the muMATH Symbolic Math System for personal computers. It contains user contributed application programs, announcements of new releases or revisions to the software, and a forum for the growing community of muLISP and muMATH users. Registered owners of muLISP can obtain a *free copy* by writing Soft Warehouse, Inc. Please include the serial number on your muLISP master diskette.

1.2 muLISP SYSTEM CONTENTS

muLISP-87 consists of the muLISP-87 Reference Manual and the following files on the master diskettes:

Executable COM files:

MULISP.COM	the muLISP interpreter
COMTOEXE.COM	a COM to EXE file conversion utility

Program Development System files:

WINDOWS.LSP	window manager
EDIT.LSP	source file editor
DEBUG.LSP	program debugger
ADDRESS.LSP	data entry program
HELP.LSP	help facility
MULISP86.LSP	muLISP-86 compatibility file

Package Library files:

COMMON.LSP	Common LISP package
IRRATNAL.LSP	irrational function package
ARRAY.LSP	array package
STRUCTUR.LSP	structure package
FLAVORS.LSP	flavors package
GRAPHICS.LSP	turtle graphics package
NECGRAPH.LSP	NEC PC-9801 graphics package
MOUSE.LSP	Microsoft Mouse package
CLOSURE.LSP	dynamic closure package

Tutorial System files:

LESSONS.LSP	Tutorial System lesson driver
MULISP1.LES	data objects and primitive functions
MULISP2.LES	defining functions using recursion
MULISP3.LES	symbols, numbers, and conses
MULISP4.LES	list processing & iterative functions
MULISP5.LES	numerical programming techniques
MULISP6.LES	implementing turtle graphics functions

Demonstration Game files:

ANIMAL.LSP	Animal game
ANIMAL.MEM	Animal game's prebuilt data base
DOCTOR.LSP	Doctor or Eliza
EIGHTS.LSP	Eights puzzle
HANOI.LSP	Tower of Hanoi puzzle
METAMIND.LSP	Metamind game

1.3 SYSTEM REQUIREMENTS

muLISP-87 runs on a variety of personal computers using either the MS-DOS or PC-DOS operating system. This section lists the minimum computer hardware and disk operating system (DOS) software required to run muLISP.

1.3.1 IBM Personal Computer

1. An IBM Personal Computer (either PC, XT, AT, Laptop, or Personal System/2) with either a monochrome monitor or a graphics display monitor.

2. A minimum of 256 Kilobytes of computer memory.

3. At least one double-sided (360K) $5^1/_4$ inch floppy disk drive or one double-sided (720K) $3^1/_2$ inch disk drive.

4. The diskette containing the PC-DOS operating system software, version 2.0 or later, that is provided with IBM Personal Computers.

1.3.2 MS-DOS Computers

1. An 8086, 8088, 80186, 80286, or 80386 based microcomputer running Microsoft's MS-DOS operating system software, version 2.0 or later.

2. A minimum of 256 Kilobytes of computer memory.

3. At least one standard disk drive capable of reading the muLISP master diskettes (contact your muLISP dealer to determine the availability of muLISP diskettes formatted for your disk drive).

1.4 MASTER DISKETTE BACKUP

As soon as possible after receipt of muLISP, make a working cop of the muLISP master diskette. This is a safeguard against the loss damage or accidental erasure of the master diskette. Once copied, th master diskette should be stored in a safe, cool place *separate* from th working diskette.

All copies of muLISP must have a label bearing the followin copyright notice:

(C) COPYRIGHT SOFT WAREHOUSE, INC., 1982, 1983, 1985, 1986, 1987.
ALL RIGHTS RESERVED WORLDWIDE.

In addition to the copyright notice, it is a good idea to put on th label the version number (i.e. 6.*nn*), the version date (i.e. *mm/dd/yy*), an the serial number as printed on your master muLISP diskette. Includ these numbers in any inquiries you make concerning muLISP.

If your computer has a diskette copy utility, use it to make you working copy of muLISP. Your computer's operating system manua should describe how to use the utility. For example, MS-DOS users ca use the DISKCOPY program described in the DOS Reference Manual.

If your computer has two or more disk drives, another way you ca make a working copy of muLISP is to perform the following steps:

1. Turn on the computer and bootstrap DOS using a system diskette.
2. Format a blank diskette using the DOS format utility.
3. Insert the newly formatted diskette in drive B.
4. Insert the muLISP master diskette in drive A.
5. Following the DOS prompt, enter the command

```
COPY A:*.* B:
```

and press the RETURN key. The system will proceed to copy th files on the master diskette to the working diskette.

6. When all the files have been copied to the working diskette, the computer will redisplay the DOS prompt.

7. Remove the master diskette from drive A and store it in a safe place.

8. Remove the working diskette from drive B and place the Soft Warehouse copyright notice on it.

1.5 THE IBM PC KEYBOARD

This section will be of interest only if you are running muLISP on an IBM PC or compatible computer. Some of the keys on the IBM PC keyboard are marked with descriptive symbols instead of spelled-out names. Throughout this manual these keys will be referred to using names printed in small capital letters. The following table gives the names, the ASCII codes, and the corresponding symbols used on the keyboard:

Key	Code	Symbol
BACKSPACE	8	<—
ESC	27	Esc
LINEFEED	10	Ctrl <-┘
RETURN	13	<-┘
RUBOUT	127	Ctrl <—
SPACE	32	space bar
TAB	9	—>|

2. muLISP OPERATING PROCEDURES

This chapter describes how to load and run muLISP, how to enter and edit commands, how to define new functions, how to load source files, and how to load and save SYS files.

2.1 SYSTEM LOAD AND RUN

Once the master diskette is safely backed up, it is a simple matter to initiate execution of muLISP. First bring up the computer's disk operating system (DOS) in the normal manner. Next, if necessary, switch the default drive to the one with the diskette containing a copy of MULISP.COM. Then, following the operating system prompt, enter the command

```
MULISP
```

and press the RETURN key. After a few seconds of load time, the muLISP header message

```
muLISP-87 xxxxxxxxx MS-DOS Version 6.nn (mm/dd/yy)
(C) Copyright Soft Warehouse Inc. 1983, 1985, 1986
All Rights Reserved Worldwide
```

is displayed on the screen. Appropriate numbers will appear for the version number (*6.nn*), month (*mm*), day (*dd*), and year (*yy*). The version number and version date, along with your software serial number, should be included in all inquiries concerning the system. The *xxxxxxxxx* in the header message indicates the type of computer system on which this version of muLISP is being run.

The muLISP prompt is displayed below the header message. The default prompt consists of a dollar sign ($) character followed by a space and the console cursor. The prompt indicates the system is ready to accept console input. The muLISP prompt string can be changed by modifying the muLISP base page (see Appendix B).

2.2 THE READ-EVAL-PRINT LOOP

Following the prompt, expressions written in list and/or dot notation can be entered by the user from the console. Multi-line expressions can be entered since an expression is not evaluated until all parentheses are balanced.

After a complete LISP expression is entered by the user and the RETURN key is pressed, muLISP *reads* the expression, *evaluates* it, and *prints* the result beginning on a new line. This read-eval-print loop is repeated indefinitely until the user enters a SYSTEM command. A SYSTEM command terminates muLISP and returns control to DOS.

The following printout of a sample muLISP session demonstrates the read eval-print loop. Comments are printed at the right margin following semicolons (the muLISP comment delimiter). These comments are for information only and need *not* be entered for the examples to work.

```
$ 'DOG                                        ;Quote a symbol
DOG

$ (+ 5 -2 4)                                  ;Add three integers
7

$ (EQUAL 'DOG 'CAT)                           ;Compare two symbols
NIL

$ (member 'dog '(cat cow dog pig))             ;Search a list
(DOG PIG)

$ (SYSTEM)                                    ;Quit muLISP, return to DOS

A>                                            ;The DOS prompt
```

Note that by default muLISP converts lower case letters to upper case as they are read. However, if the control variable *READ-UPCASE* is set to NIL, lower case leters are *not* converted to upper case (see *READ-UPCASE* in Chapter 5).

As a programming convenience, the last three expressions read by the read-eval-print loop are assigned to variables: the last expression to +, the second from last to ++, and the third from last to +++. Also the evaluated result of the last three expressions read are assigned to variables: the last result to *, the second from last to **, and the third from last to ***. For example:

```
$ (APPEND '(TOM SAM JOAN) '(ANN JOE SUE))
(TOM SAM JOAN ANN JOE SUE)

$ +
(APPEND '(TOM SAM JOAN) '(ANN JOE SUE))

$ **
(TOM SAM JOAN ANN JOE SUE)
```

The default read-eval-print loop is sometimes called an eval-LISP driver loop. For some applications it may be advantageous to define your own read-eval-print loop. This can be accomplished by assigning the control variable DRIVER to your own read-eval-print loop function (see the User Interface Functions section of Chapter 5).

2.3 LINE EDITING PROCEDURES

muLISP provides a powerful line editor that enables you to edit the previously entered line and the current line of keyboard input. To enter a line of text, simply type in the characters and then press the RETURN key.

Editing of the text is done by entering control characters to perform various functions. There is a control character associated with each letter of the alphabet. A control character is entered by typing the associated letter while holding down the CTRL key. This table summarizes the effect of the control characters on the line of text.

Control Key	Alternate	IBM Key	Effect
CTRL-A		CTRL <—	move left a word
CTRL-D		—>	move right a character
CTRL-F		CTRL —>	move right a word
CTRL-G		Del	delete character under cursor
CTRL-H	BACKSPACE	<—	delete character left of cursor
CTRL-I	TAB	—>|	move right to next tab stop
CTRL-J	LINEFEED	CTRL <┘	enter line of text
CTRL-M	RETURN	<┘	enter line of text
CTRL-P			escape for next char entered
CTRL-Q D			move to right end of line
CTRL-Q S			move to left end of line
CTRL-Q Y			delete right end of line
CTRL-Q RUBOUT			delete left end of line
CTRL-S		<—	move left a character
CTRL-T			delete word under cursor
CTRL-U		F3	undelete last deleted line
CTRL-V		Ins	insert/replace mode toggle
CTRL-Y			delete line of text
CTRL-_	RUBOUT	CTRL <—	delete character left of cursor
CTRL-[	ESC	Esc	escape current break level

These line editor commands are similar to those provided by the full-screen muLISP Editor (see Chapter 7). The line editor commands can be reassigned to different control keys by modifying the muLISP Base Page (see Chapter 9 and Appendix B).

Note that if muLISP is running on an IBM PC or compatible computer, the cursor control keys (i.e. left arrow, right arrow, Home, End, Del, Insert, etc.) at the right end of the keyboard are also functional.

muLISP output to the console can be "frozen" and then restarted using CTRL-S as a toggle. This is useful to prevent the scrolling of text off the screen before it can be read.

2.4 PROGRAMMING FUNDAMENTALS

Although it does not fully take advantage of the capabilities of muLISP, the following is a perfectly acceptable recursive definition of the factorial function:

```
(DEFUN FACTORIAL (N)
  (COND
    ((ZEROP N) 1)
    (T (* N (FACTORIAL (SUB1 N))))  ) )
```

muLISP-87 incorporates numerous extensions to LISP that significantly increase program readability and execution speed, while substantially decreasing the storage requirements for function definitions. Here are a few of the significant extensions:

1. When a new symbol is read or generated by muLISP, its value is automatically set to itself. This self-referencing of symbols is called auto-quoting. It often eliminates the need for quoting symbols in definitions, thereby improving performance and saving space.

2. The COND function has been generalized so all the expressions following a nonNIL predicate are evaluated in turn. The value returned by this extended COND is the value of the last expression.

3. The evaluation of function bodies has been extended to use an implicit PROGN so that function bodies can contain more than a single task. muLISP also recognizes implicit CONDs in function definitions, thus greatly reducing the need for explicit CONDs in definitions.

Comparing the following recursive definition of the factorial function with the previous one illustrates some of the extended capabilities of muLISP:

```
(DEFUN FACTORIAL (N)
  ((ZEROP N) 1)
  (* N (FACTORIAL (SUB1 N))) )
```

With the principal exception of the unstructured PROG and GO program control constructs, LISP is an exemplary applicative and structured language. muLISP has two features which dispose of any need for such unstructured control features:

1. A powerful multiple-exit LOOP function permits the programming of nonrecursive loops. This is achieved without the use of the totally unstructured GO feature.

2. Arguments in a function's formal argument list that are in excess of the actual number of arguments used in a call to the function are simply bound to NIL. These excess arguments are available for use as local variables within the function. Also, in-line Lambdas and the LET control construct (see the Control Constructs section of Chapter 5) can be used to introduce local variables in functions. For these two reasons, muLISP does not need PROG to establish local variables.

Therefore the PROG and GO control constructs are not primitively defined in muLISP. However if necessary for compatibility, the source file COMMON.LSP does provide an interpreter for them.

Instead of requiring numerous explicit control constructs, muLISP programs can be guided primarily by the tree structure of function bodies. Program control using implicit PROGNs and implicit CONDs results in function definitions whose structures and meanings are more transparent, require less memory to store, and are much more efficient. The Control Constructs section of Chapter 5 gives a detailed explanation of these concepts.

As an example of how to replace explicit control constucts with implicit PROGNs and CONDs, consider the following iterative definitions of the FACTORIAL function:

```
(DEFUN FACTORIAL (N)                    ;PROG definition
  (PROG (M)
    (SETQ M 1)
  A (COND
      ((ZEROP N) (RETURN M)) )
    (SETQ M (* M N))
    (SETQ N (SUB1 N))
    (GO A) ) )

(DEFUN FACTORIAL (N M)                  ;muLISP definition
  (SETQ M 1)
  (LOOP
    ((ZEROP N) M)
    (SETQ M (* M N))
    (SETQ N (SUB1 N)) ) )
```

The first is a conventional nonrecursive definition (Note: It is *not* a workable muLISP function definition). The second is an *equivalent* muLISP definition that is also nonrecursive. You can decide for yourself which definition is more elegant and structured. It is interesting to note that the muLISP definition requires only 28 conses; whereas, the conventional one requires 37 conses. This is a significant space saving for a LISP system running in a limited address space.

After defining the FACTORIAL function, you can generate some really large factorials in a hurry using muLISP. For example:

```
$ (FACTORIAL 100)
93326215443944152681699238856266700490715968264381
62146859296389521759999322991560894146397615651828
62536979208272237582511852109168640000000000000000
00000000
```

2.5 READING SOURCE FILES

muLISP source files usually contain function definitions, variable assignments, and other commands. Numerous source files are included on the muLISP master diskette (see Chapter 1). If *filename* specifies the drive, directory, and name of a source file, the LOAD command

(LOAD '*filename*)

reads in and processes *filename* making its facilities available for use. If *filename* is not found, the LOAD command returns NIL. If *filename* does not specify a file name extension, the file name extension "LSP" is used. If *filename* does not specify a drive, the MS-DOS default drive is used. For example, the command

```
$ (LOAD 'ANIMAL)
```

loads the source file ANIMAL.LSP from the default drive.

An alternative way to load a source file is to include the name of the file in the DOS command used to start muLISP. This causes the source file to be loaded immediately after the muLISP interpreter is loaded. For example, the DOS command

```
B>A:MULISP ANIMAL.LSP
```

loads the source file ANIMAL.LSP from the default drive after muLISP is loaded from drive A. The file name extension "LSP" can be omitted from the source file name; however if there is also a SYS file with the same name, the SYS file is loaded instead of the source file (see Section 2.6).

Note that source files can be chain-loaded by using LOAD commands in the source files. See the File System Primitives section of Chapter 5 for more details about the function LOAD.

2.6 ENVIRONMENT SYS FILES

The muLISP environment is the current state of the system. It consists of all the currently active muLISP data structures, variable values, property values, and function definitions. The SAVE command

(SAVE '*filename*)

saves the current muLISP environment as a SYS file on the drive and directory specified by *filename*. This environment can be quickly restored by the LOAD command

(LOAD '*filename*)

Note that the file name given to LOAD and SAVE commands need not include the file name extension "SYS". For example, the SAVE command

```
$ (SAVE 'B:WHALE)
```

saves the current environment as the SYS file WHALE.SYS on drive B. This environment can be restored at any time by the LOAD command

```
$ (LOAD 'B:WHALE)
```

If there is no SYS file or source file named WHALE on drive B, the LOAD command returns NIL. However, if the SYS file is found, the LOAD command does not return a value but immediately begins execution in the new environment.

Note that the environment present at the time the LOAD command is issued is *replaced* with that stored in the SYS file. Thus unlike loading source files, loading SYS files is *not* cumulative. Instead the new SYS file's environment destroys the existing environment, including all function definitions.

An alternative way to load a SYS file is to include the name of the file in the DOS command used to start muLISP. This causes the SYS file to be loaded immediately after the muLISP interpreter is loaded. For example, the DOS command

```
A>MULISP B:WHALE
```

loads the SYS file WHALE.SYS from drive B after muLISP is loaded from the default drive. When an attempt is made to load a source or SYS file using a DOS command and the file is not found, the warning message "File not found" is displayed after the muLISP header message.

When a SYS file is loaded using either of the above methods, the environment is restored exactly as it was at the time of the save. After loading the SYS file, the amount of memory allocated to each of the various data spaces is adjusted according to the computer's current memory size. This means that the current memory size does not necessarily have to be the same as when the SYS file was created. However if there is insufficient memory to store the SYS file's environment, an "Insufficient Memory, Aborting" error will occur and muLISP will be terminated.

A SYS file can be useful in a variety of ways. For instance, an interactive session can be continued at a later time, intermediate results of a session can be backed up as insurance against computer failure, or a set of function definitions developed interactively can be preserved.

The development of a large program is usually done by creating source files rather than SYS files. However, once the program has been perfected, a SYS file containing the source files can be saved so the program can be loaded at any future time both quickly and conveniently.

(This page intentionally left blank)

3. LEARNING muLISP

This chapter describes how to load and play the demonstration game programs and how to start the muLISP Tutorial System.

3.1 DEMONSTRATION PROGRAMS

Several demonstration game programs are included with the muLISP system. These programs demonstrate numerous muLISP programming techniques such as manipulating data structures and providing a friendly user interface. Each program can be loaded by issuing a LOAD command following the muLISP prompt. For example, to load the Animal game from the disk in drive B, issue the command

```
$ (LOAD 'B:ANIMAL)
```

ANIMAL.LSP Think of an animal. The program will try to guess the animal by asking simple yes/no questions. The Animal Game is an excellent example of the use of a data base set up as a binary discrimination net. If the program fails to guess the animal, it asks you how to distinguish the animal it guessed from the animal you were thinking of, and extends its data base to include your animal. The more you play, the "smarter" the program becomes! The animal data base can be saved as a text file for use in a later session. ANIMAL.MEM is an optional, prebuilt database for the animal game containing over 100 animals. If the player requests it, this data base is automatically loaded by ANIMAL.LSP.

DOCTOR.LSP This is the muLISP version of the Eliza (sometimes called Doctor) program written by Joseph Weizenbaum of MIT. It turns your computer into an amateurish, but persistent, psychologist. It mimics the Rogerian school of psychology where the doctor asks leading questions, letting patients resolve their own problems.

The "P:" (patient) prompt indicates that the "doctor" is waiting for you to comment on your problem. Don't forget to terminate your sentence with a period, question mark, or exclamation point. The

"D." (doctor) herald precedes the doctor's responses to your statements. To exit the program, just type "Goodbye."

Most of the program consists of a script that contains the keywords, patterns, and responses used by the doctor. The script can be elaborated upon or replaced with a script on an entirely different subject.

EIGHTS.LSP The Eights Puzzle consists of eight square "tiles" numbered 1 through 8 in a square "tray" that can accommodate nine tiles. The program asks the user to place the tiles on the tray in a scrambled order. Using a heuristic search, the program finds and displays a sequence of moves that rearranges the tiles into a standard order. A move consists of sliding a tile vertically or horizontally into the adjacent empty space.

The standard order can only be reached from half the initial positions. Before beginning the search, the program uses a quick, direct test to determine if the standard order is reachable. The test is based on the number of permutations between the initial and final positions.

HANOI.LSP This program graphically demonstrates the recursive solution to the Tower of Hanoi Puzzle. There are three pegs, one of which has a stack of rings of successively smaller diameter from bottom to top (i.e. the smallest on the top). The objective is to transfer all the rings to one of the other pegs so they will be in the same order. Only one ring may be moved at a time, and at no time may a larger ring lie on a smaller ring.

The program begins by asking how many rings you want to move, whether you want to run the program at normal speed or high speed, whether or not you want sound effects, and whether or not you are using a color monitor. In normal speed, the program simulates ring movement by showing intermediate positions of the rings and using sound effects.

METAMIND.LSP Metamind is similar to the popular Master Mind[tm] board game by Invicta. One player, the "code maker", chooses a secret code consisting of a sequence of four (not necessarily different) colors. The other player, the "code breaker", tries to break the code in a minimal number of guesses. In response to the code breaker's guess as to the code, the code maker states the number of exact matches (i.e. of the correct color in the correct position), and the number of correct colors that are in the wrong position. The computer can be either the code maker or breaker.

HELP.LSP This file provides a help facility for the large number of muLISP primitives. After it has been loaded, commands of the form (HELP *name*) or (? *name*) display *name*'s arguments, primitive type, and the page number in Chapter 5 containing the detailed description of *name*.

3.2 muLISP TUTORIAL SYSTEM

The muLISP Tutorial System is an interactive, on-line system designed to teach the fundamentals of LISP programming. The source file LESSONS.LSP contains the muLISP program that presents the lessons. Each of the LESson files, named MULISPn.LES, contain the actual text and examples for each lesson.

To run the Tutorial System you will need to start up muLISP as described in Section 2.1 and then insert a diskette containing LESSONS.LSP and the LES files in the logged drive. After the muLISP dollar sign prompt, issue the LOAD command

```
$ (LOAD 'LESSONS)
```

After the lesson program is loaded, it will display a menu listing the available lessons with a brief indication of their contents. To start with the first lesson, simply press the SPACE BAR. The screen will clear and the lesson will begin. From this point on the lessons should be self explanatory.

The lessons should be taken in order the first time through.

4. THE muLISP INTERPRETER

This chapter describes the primitive muLISP data structures, the memory management system, the break package, the console interrupts, and the error messages.

4.1 DATA STRUCTURES

Symbols, numbers, and conses make up the primitive muLISP data objects. Using the recognizer functions, the type of a data object can be determined. Each data object of a given type consists of a fixed number of pointer cells. These cells can point to (i.e. contain the memory address of) other data objects or to special purpose data structures such as character strings or binary number vectors. The set of all data objects form an interconnected network of pointers called the muLISP data space.

All three types of data objects have as a minimum a car cell and a cdr cell (pronounced "could-er"). These two cells can only point to other objects within the muLISP data space. Thus, the data space is "closed" in the sense that by following car and cdr cell pointers you cannot leave the space. The advantage of this closure is a simpler and more logical muLISP evaluation mechanism. For instance, it eliminates the need for time consuming run-time type checking by the functions that follow or manipulate these two pointer cells.

4.1.1 Symbols

A symbol is a recognizable muLISP data object consisting of four pointer cells:

Value	Property	Function	Pname

Symbols are uniquely stored in the sense that no two symbols in the system can have identical print names. The four cells are used as follows:

Value cell This cell contains a pointer to the symbol's current value. When a symbol is created, its value cell is set to the symbol itself. This automatic self-referencing of symbols is called auto-quoting. The assignment functions are used to change the contents of a symbol's value cell. The value cells of a function's formal parameters are temporarily reassigned when the function is called, and then restored to their original value when the function is exited.

Property list cell This cell contains a pointer to the symbol's property list. This list is used and modified by the property and flag functions. Flags on a property list can be distinguished by their being atomic elements of the list (symbols or numbers). A property consists of an indicator key consed to a property value. When a symbol is created, its property list is set to NIL, indicating that no flags or properties are present.

Function cell This cell contains a pointer to the symbol's definition. Function definitions are either machine language routines or compiled D-code. When a symbol is applied, the function cell is used by the muLISP evaluation mechanism to locate the symbol's definition. The function definition primitives are used to retrieve and modify function definitions. When a symbol is created, its function cell points to an undefined function trap.

Pname cell This cell contains a pointer to the symbol's print name string. The string consists of two bytes that store the length of the string followed by a series of bytes that contain the ASCII characters used to print the symbol. Thus strings are "limited" in size to less than 65536 characters.

Access to the pname cell and string is restricted to the I/O and string functions. When a print name string is read in or generated by a string function, a hashing algorithm is used to determine if a symbol already exists with that print name. If so, the existing symbol is used. If not, another symbol is created using the new print name. Once created, a symbol's print name string cannot be modified.

4.1.2 Numbers

A number is a recognizable muLISP data object consisting of four pointer cells:

Identity	Sign	Length	Vector

Numbers can be further subdivided into integers and ratios. Internally, integers are further subdivided into small integers (those less than 65536 in magnitude) and big integers. Small integers are stored uniquely. Ratios and big integers are *not* stored uniquely, so numbers with the same numerical value can coexist in the system. Except for the function EQ, the distinction between small and big integers does not affect the muLISP programmer. The four cells of a number are used as follows:

Identity cell This cell contains a pointer to the number itself. Thus numbers do not have to be quoted, since they evaluate to themselves. This cell can be accessed as the car of a number but it cannot be changed by a muLISP program.

Sign cell This cell determines a number's sign and whether it is a small integer, a big integer, or a ratio. The cell contains a pointer to one of six symbols in accordance with the following table:

	Positive	Negative
Small Integer	NIL	T
Big-Integer	LAMBDA	NLAMBDA
Ratio	MACRO	SPECIAL

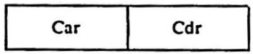
The contents of the sign cell can be accessed as the cdr of a number but it cannot be changed by a muLISP program.

Length cell If the number is a small integer, this cell contains the magnitude of the integer. If the number is a big integer, this cell contains the word length of the number's vector. If the number is a ratio, this cell contains a pointer to the numerator of the ratio, which must be an integer.

Vector cell If the number is a small integer, this cell contains a hash link (i.e. a pointer to another small integer) or a 1. If the number is a big integer, this cell contains a pointer to the least significant word of the number's vector. A number vector is a contiguous series of words in memory used to store the unsigned magnitude of a muLISP number in binary. The words of a number vector are stored in reverse order (i.e. more significant words are stored in successively lower memory addresses). If the number is a ratio, this cell contains a pointer to the denominator of the ratio, which must be a positive integer.

4.1.3 Conses

A cons is a recognizable muLISP data object consisting of two pointer cells:



Binary trees are the primary data structures in muLISP. Internally they are implemented as a network of cell pairs called conses. Each cons consists of a car cell and a cdr cell. As mentioned earlier, the cells of a cons can only point to bona fide muLISP data objects: either symbols, numbers, or conses.

Conses are often called dotted-pairs, because of the dot notation sometimes used to represent LISP conses. If X and Y are data objects, the expression

```
(X . Y)
```

denotes a cons whose car cell points to X and whose cdr cell points to Y. Although cumbersome, dot notation is capable of showing the structure of any LISP data object.

In most applications it is more convenient to think of a data structure as a linear list rather than as a deeply nested binary tree. Therefore data structures are usually expressed in list notation. The list

```
(X1 X2 ... Xn)
```

represents a chain of conses linked through their cdr cells and whose car cells point to the elements of the list. To be a valid list, the cdr cell of the last cons of the list must point to the atom NIL. The object above expressed in dot notation is

```
(X1 . (X2 . (... . (Xn . NIL)...)))
```

The function READ accepts both dot and list notation. In list notation, the null or empty list is denoted by NIL. Therefore, when "( )" is read by READ, NIL is returned.

The functions WRITE, PRIN1, PRINC, and PRINT use list notation to the maximum extent possible when printing muLISP data objects. For example,

```
(DOG . (CAT . COW))
```

is displayed by WRITE, PRIN1, PRINC, and PRINT as

```
(DOG CAT . COW)
```

4.2 MEMORY MANAGEMENT

Dynamic, automatic memory management gives LISP much of its inherent power. This frees the programmer from having to allocate, in advance, sufficient memory for a problem. Such allocation before run-time is difficult, if not impossible, for most AI applications since the problems are generally not of predetermined size. During the execution of a program, all of the memory not actually required by the program should be available for the creation of new data structures. This is achieved in muLISP in phases, as described in the following subsections.

4.2.1 Initial Data Space Partition

During the initialization phase of muLISP, the amount of read/write memory available to the system is computed. Memory is then partitioned into four distinct data spaces. The proportion of memory initially assigned to each space is based on the relative demands made on the spaces by typical muLISP application programs. The following are the names of the four data spaces and their respective uses:

Atom Space provides space for the four pointer cells required for each symbol and number.

Vector Space provides space for each symbol's print name string and for each big integer's binary number vector.

Pointer Space provides space for the two pointer cells required for each cons and the D-code (distilled code) required for each function definition. Since conses are the primary muLISP data structure, the pointer space is usually the largest data space.

Stack Space provides space for the control stack and the variable stack. The two stacks are based at opposite ends of the stack space.

If muLISP is running on a computer based on one of the Intel 8086 family of microprocessors, the address space of the computer is divided into 64K (Kilobyte) segments. Unfortunately this imposes a somewhat arbitrary limit on the size of each muLISP data space. Thus, the space available for the vector space is 128K total: 64K for print name strings and 64K for number vectors. The space available for the pointer cells

making up data objects (i.e. symbols, numbers, and conses) is 256K. The space available for primitive and user-defined machine code is 64K. The space available for each D-code space and the stacks is 64K. There can be up to six D-code spaces (see the description of NEW-CODE-SPACE in Chapter 5).

In short, muLISP can take advantage of up to 640K of computer memory. However, it may happen that one of the muLISP data spaces becomes exhausted even though other spaces still have some free space. If this situation should occur during the execution of a program, muLISP generates a "Memory Full" error break (see Section 4.5).

4.2.2 Garbage Collection

The nature of stack operations makes management of the stack space automatic and continuous. However, management of the remaining three spaces requires explicit recycling of data objects that are no longer needed. New data objects are constantly being *created* during the execution of a muLISP program, while others are implicitly *discarded* when they are no longer referenced. When the creation process uses up the available resources in any one of the data spaces, a garbage collection is performed on *all* the spaces. Storage space vacated by discarded data structures is reclaimed so the program can continue.

A two-pass, mark-sweep garbage collection algorithm is used. The *first pass* consists of marking all active data objects. These are the objects accessible by means of chaining through pointer cells beginning from the value and property list cells of all system symbols, or from the variable stack, or from distilled code. Unreferenced, auto-quoted symbols that have no properties and have no current function definition are not marked. Such symbols are automatically removed from the object list during the second pass.

During the *second pass* of the garbage collection, all the marked data objects are collected and compacted into one end of their respective data spaces. This leaves the remainder of each data space available for newly created objects.

4.2.3 Data Space Reallocation

After a garbage collection one or more of the four muLISP data spaces may have an insufficient amount of free storage to permit an application program to continue execution, despite the fact that one or more of the other data spaces may have abundant free space. If this situation occurs, a data space reallocation routine is called to move the partitions dividing the spaces, giving more memory to the exhausted space or spaces. (However, the limits explained earlier on the size of each data space must be observed by the reallocator.) Thus muLISP can respond to changing demands placed on the various data spaces by application programs.

Although garbage collection and data space reallocation are automatic, their occurrence is not entirely invisible to the user since they cause a short pause in the execution of a program. If muLISP is running on an IBM PC, the blinking cursor temporarily changes to a full block during collections and reallocations.

The exact amount of time required for a collection or reallocation depends on the amount of muLISP data currently in the system. A garbage collection usually takes less than a second. Similarly, a data space reallocation usually takes less than a second. Normally this is of no concern to the user, but it should be considered in the design of real time systems using muLISP.

A phenomenon known as thrashing occurs when the system is forced to spend an inordinate amount of time garbage collecting for a very small return of data space. The symptom of thrashing is greatly increased execution time for a given task. This problem can sometimes be resolved by increasing the computer's memory size (up to 640K) and/or modifying the program to reduce its storage requirements.

4.3 THE BREAK PACKAGE

The muLISP break package is called by the keyboard interrupt handler (see Section 4.4) and by the error trap handlers (see Section 4.5). When a break occurs, the option prompt

```
Continue, Break, Abort, Top-level, Restart, System
```

is displayed after the interrupt or error message. The system then waits for the user to select one of the options by entering the first letter of its name (C, B, A, T, R, or S). Note that the options are listed in order of increasingly drastic action.

Continue returns control to the program that invoked the break routine. If the break was caused by a console interrupt, execution continues as if the break had not occurred. If the break was caused by an error trap, the value passed to the break by the error handler is returned as the value of the errant function.

Break temporarily suspends program execution and calls the next lower-level read-eval-print driver loop. This allows the user to examine and/or change the current muLISP environment before continuing. To exit the break and resume program execution, press the ESC key following the dollar sign prompt. See the User Interface Section of Chapter 5 for a detailed discussion of the read-eval-print driver loop.

Abort aborts program execution, restores the original value to the formal arguments that are stored on the variable stack, and returns control to the current-level read-eval-print loop. Function definitions, property values, and global variable values are left unchanged.

Top-level aborts program execution, restores the original value to the formal arguments that are stored on the variable stack, makes the console the Standard Input Source (SIS) and Standard Output Sink (SOS), and returns control to the top-level read-eval-print loop. Function definitions, property values, and global variable values are left unchanged.

Restart closes all open files, abandons the current muLISP environment, and initiates a fresh muLISP system. All variable bindings, user-defined functions, and property values in the current muLISP environment are destroyed.

System closes all open files, terminates execution of muLISP, and returns control to the host operating system.

4.4 CONSOLE INTERRUPTS

At any time during the execution of a program, a user initiated console interrupt will break program execution. This may be necessary to stop a "runaway" or nonterminating program and return control to the console.

A console interrupt is initiated by pressing the ESC key on the console keyboard. If using a keyboard without an ESC key, the interrupt character can be generated by typing the left bracket key ([) while holding down the CTRL key. If that does not work, the character used to generate keyboard interrupts can be changed by modifying the Read Table in the muLISP Base Page (see Appendix B).

When a console interrupt break occurs, the message

```
Console Interrupt Break: NIL
```

is displayed on the console screen. This is followed on the next line by the break option prompt (see Section 4.3). The user can then decide how to proceed.

If for some reason there is no response to a keyboard interrupt, it is always possible to interrupt the system by resetting the computer. However, a computer reset done in the middle of an evaluation leaves muLISP in an unpredictable state, making it impossible to restore the muLISP environment present at the time of the reset.

4.5 ERROR MESSAGES

When muLISP detects an error condition, it generates an error break by calling the function BREAK with the form that provoked the error and an appropriate error message. BREAK displays the error message and the errant form, suspends program execution, and provides the user with options on how to proceed (see Section 4.3).

Conditions that cause error breaks include invalid arguments, undefined functions, space exhaustion, syntax errors, and end-of-file reads. The following is a summary of all the muLISP error breaks except those specific to the muLISP compiler (see Chapter 8 for those):

Dimension Error indicates that the function MAKE-ARRAY was called with neither a positive integer nor a list of positive integers for the dimensions of the array, or that it was called with an initial contents expression incompatible in length with the dimensions of the array. See Section 6.1 for details.

Disk Full indicates that there is insufficient space to store the data being written to a disk file. If this situation should occur, a throw is made to the label "Disk Full" with a value of NIL. If this throw is not caught by an application program, the muLISP top-level loop (see Section 5.19) catches the throw and generates an error break. Since the file remains open, it may be possible to delete other files from the full diskette (using the function DELETE-FILE), then continue writing to the file.

End-Of-File indicates that an attempt was made to read from beyond the end of an input file or from "holes" within the input file (see the FILE-WRITE-POSITION function in Chapter 5 for an explanation of holes in randomly written files). Following the "End-Of-File" error message, the full pathname of the input file is displayed.

Exact Mode indicates that exact mode arithmetic is in effect (i.e. the control variable *EXACT-MODE* is neither NIL nor the symbol RATIONAL) and an irrational numerical function (see Section 5.12.5) was called upon to generate an approximate result.

File not found indicates that the source and/or SYS file given on the operating system command line that initiated muLISP could not be found, or is a SYS file created under a muLISP with a different version number *or* version date. A SYS file can only be loaded under the same version of muLISP that was used to create the file. Chapter 2 describes how to load files using the operating system command line.

Source and SYS files can also be loaded from within muLISP using a LOAD command. When a LOAD command is issued from within muLISP and the file is not found, the command returns NIL instead of displaying the "File not found" message.

Include Error indicates that the macro DEFSTRUCT was called upon to include a named structure in an unnamed structure, or to include an unnamed structure in a named structure. See Section 6.2 for details.

Incompatible Definition indicates that the macro DEFSTRUCT was called upon to create a new definition for a structure function that was incompatible with the existing definition. See Section 6.2 for details.

Insufficient Arguments indicates that the function MAX, MIN, -, /, ADD1, SUB1, LCM, ABS, SIGNUM, FLOOR, CEILING, TRUNCATE, ROUND, MOD, REM, DIVIDE, NUMERATOR, DENOMINATOR, LOGNOT, INTEGER-LENGTH, or SHIFT was called without any arguments.

Insufficient Memory, Aborting indicates that there is an insufficient amount of memory in which to load and run a muLISP environment. muLISP is aborted and control returns to the host operating system.

Note that a muLISP environment saved in a SYS file can be loaded into a computer having less memory than the computer on which it was created. The insufficient memory error occurs only if the computer on which the SYS file is being loaded does not have enough memory to store the muLISP environment. The only way to load the SYS file is to obtain more computer memory.

Invalid Argument indicates that an irrational or transcendental numerical function (see Section 5.12.5) was called with an argument for which the function has no real number value.

Invalid Place Form indicates that the macro SETF, INCF, or DECF was not called with a valid place form (see SETF in Section 5.7).

Keyword Not Handled indicates that the macro MAKE-INSTANCE was called with keywords that were not handled by the inherited and included flavors of the flavor. See Section 6.3 for details.

Memory Full indicates that there is insufficient memory to continue execution of the current muLISP program. If this situation should occur, a throw is made to the label "Memory Full" with a value of NIL. If this throw is not caught by an application program, the muLISP top-level loop (see Section 5.19) catches the throw and generates an error break.

Normally, the memory management system provides sufficient storage in each data space to continuously satisfy the demands of muLISP programs. This error occurs if the memory requirements for storing data objects exhaust all available resources. Statistics in the following form are displayed along with the error message:

GC: NNNN AAAA/AAAA VVVV/VVVV PPPP/PPPP SSSS/SSSS TTTT/TTTT

The hexadecimal numbers following "GC:" (garbage collection) show the amount of space remaining in each of the four major data spaces. Thus, the data space responsible for the error can be determined. (*Note*: If muLISP is running on a computer based on one of the Intel 8086 family of microprocessors, the pointer space is divided into two subregions, either one of which can become exhausted before the other. Therefore, a Memory Space Exhausted error can occur even though the statistics show free space in all four data spaces.) See the Memory Management Functions section of Chapter 5 for a detailed explanation of the meaning of these statistics.

Missing Required Flavor indicates that the macro MAKE-INSTANCE was called upon to create an instance of a flavor that did not include all the required flavors. See Section 6.3 for details.

Noninteger Argument indicates that the function LOGAND, LOGIOR, LOGXOR, LOGNOT, SHIFT, or INTEGER-LENGTH was called with a noninteger argument.

Nonnumeric Argument indicates that the function =, /=, <, >, <=, >=, MAX, MIN, +, -, *, /, ADD1, SUB1, INCQ, DECQ, GCD, LCM, ABS, SIGNUM, NUMERATOR, DENOMINATOR, FLOOR, CEILING, TRUNCATE, ROUND, MOD, REM, or DIVIDE was called with a nonnumeric argument.

Nonsymbolic Argument indicates that SET, SETQ, PSETQ, POP, PUSH, INCQ, DECQ, or DEFSTRUCT was called with a nonsymbolic argument.

Syntax Error indicates that the function READ has read either an unmatched right parenthesis (i.e. too many right parentheses) or an error in dot notation, for example (A .) or (A B . C D). Since this error break is generated by the right parenthesis and comma character macros, redefining these character macros enables the error to be handled by an application program.

Unclaimed Message indicates that the macro SEND was called upon to send a message to an instance of a flavor that did not have a method to handle the message. See Section 6.3 for details.

Undefined Function indicates that an attempt was made to apply a symbol having no function definition. Following the error message is the form that made the call on the undefined function. A common course of action when confronted with this error is to select the Break option, appropriately define the undefined symbol, and then continue with the original program by issuing the command (RETURN (EVAL BREAK)).

Undefined Structure indicates that the macro DEFSTRUCT was called upon to include an undefined structure in a structure. See Section 6.2 for details.

Unnamed Structure indicates that the macro DEFSTRUCT was called upon to define a predicate for an unnamed structure. See Section 6.2 for details.

Unsupported Option indicates that the macro DEFSTRUCT was called with an option not supported by the muLISP structure package. See Section 6.2 for details.

Zero Divide indicates that the function /, FLOOR, CEILING, TRUNCATE, ROUND, MOD, REM, or DIVIDE was called with a divisor of zero.

5. THE muLISP PRIMITIVES

This chapter describes the functions, special forms, and control variables primitively defined in the muLISP machine language kernel, and the functions and macros defined in the library source files COMMON.LSP and IRRATNAL.LSP. Chapter 6 describes the functions, macros, and control variables defined in the other source files that make the muLISP Library.

The functions, control variables, and/or macros that make up a muLISP application program must ultimately be defined in terms of these primitives. The chapter is subdivided into the categories commonly used to characterize LISP primitives.

The title line of each primitive indicates its name, its arguments or initial value, and its type. The primitive name is printed in bold face at the left margin. If the primitive is a function, its name is followed by a list, enclosed in square brackets, of zero or more names, separated by commas, representing each of its arguments. The word "Function" is printed at the right margin.

If the primitive is a special form, its name is followed by a list, enclosed in square brackets, of names representing each of its arguments. The phrase "Special form" is printed at the right margin.

If the primitive is a macro, its name is followed by a list, enclosed in square brackets, of names representing each of its arguments. The word "Macro" is printed at the right margin.

If the primitive is a control variable, its name is followed by a colon and its default value. The phrase "Control variable" is printed at the right margin.

If the primitive is *not* defined in the machine language kernel but in a library source file, the primitive type printed at the right margin is preceded by the word "Library". These primitives are only available after loading the source file containing the primitive by a command of the form

(LOAD '*filename*)

where *filename* specifies the drive, directory, and name of the source file (either COMMON.LSP or IRRATNAL.LSP).

Following the title line is a natural language description of the primitive. For a function or special form this includes a description of the arguments it accepts, the value it returns, and any side-effects it can cause. For a control variable this includes a description of the effect it has on the muLISP environment. If appropriate there may also be a discussion of subtle points and common uses of the primitive.

If appropriate, following the natural language description of a function or a special form is a formal definition of the primitive, written in muLISP. Functions are defined using the function defining special form DEFUN. Special forms are defined using the macro defining special form DEFMACRO.

Occasionally auxiliary functions are introduced to aid in the formal definition of a primitive. The names of auxiliary functions are in lower case, indicating that they are not accessible to the user. When an irreducible, primitive concept is introduced in a formal definition, the concept is described in English using lower case letters.

Finally and if appropriate, a sample muLISP session is shown that illustrates the use of a primitive. If the expression following the muLISP dollar sign ($) prompt character is entered by the user, muLISP will evaluate the expression and return the result exactly as shown on the next line of the example. Comments concerning an example are shown to the right of the example following a semicolon, which is the muLISP comment delimiter.

5.1 SELECTOR AND SEARCHER FUNCTIONS

The primary muLISP data structure is the binary tree made up of conses (branches) and terminated by atoms (leaves). The fundamental selector functions CAR and CDR respectively return the car (left) and cdr (right) branches of trees. Although successive applications of CAR and CDR are sufficient to access any part of a tree, additional selectors have been primitively defined for efficiency and convenience.

CAR [*object*] Function

(CAR *object*) returns the car of *object* (i.e. the data object pointed to by the car cell of *object*).

The correct interpretation for the car of an object depends on whether the object is an atom, and if not, whether it is being regarded as a list or a binary tree. If the object is an atom, its car cell is the current value of the atom. If the object is a list, its car cell is the first element of the list. If the object is a binary tree, its car cell is the left or car branch of the tree.

```
(DEFUN CAR (OBJ)
  (the contents of the car cell of OBJ) )

$ (CAR '(A B C D E))                    ;Car of list
A

$ (CAR '((A . B) . C))                  ;Car of binary tree
(A . B)

$ (SETQ NUM 7)                          ;Set NUM to 7
7

$ (CAR 'NUM)                            ;Car (i.e. value) of symbol
7
```

CDR [*object*] Function

(CDR *object*) returns the cdr of *object* (i.e. the data object pointed to by the cdr cell of *object*).

The correct interpretation for the cdr of an object depends on whether the object is an atom, a list, or a binary tree. If the object is a symbol, its cdr cell is the property list of the symbol. If the object is a number, its cdr cell indicates the sign and type of the number (see Chapter 4 for details). If the object is a list, its cdr cell is the tail of the list (i.e. everything but the first element of the list). If the object is a binary tree, its cdr cell is the right or cdr branch of the tree.

```
(DEFUN CDR (OBJ)
  (the contents of the cdr cell of OBJ) )

$ (CDR '(A B C D E))                    ;Cdr of list
(B C D E)

$ (CDR '((A . B) . C))                  ;Cdr of tree
C

$ (PUT 'JOE 'SEX 'MALE)          ;Put sex property on JOE
MALE

$ (PUT 'JOE 'AGE 33)             ;Put age property on JOE
33

$ (CDR 'JOE)                ;Cdr (i.e. property list) of symbol
((AGE . 33) (SEX . MALE))
```

CAAR [*object*] Function
CDAR [*object*] Function
CADR [*object*] Function
CDDR [*object*] Function

CAAAR [*object*] Function
CDAAR [*object*] Function
CADAR [*object*] Function
CDDAR [*object*] Function
CAADR [*object*] Function
CDADR [*object*] Function
CADDR [*object*] Function
CDDDR [*object*] Function

CAAAAR [*object*] Function
CDAAAR [*object*] Function
CADAAR [*object*] Function
CDDAAR [*object*] Function
CAADAR [*object*] Function
CDADAR [*object*] Function
CADDAR [*object*] Function
CDDDAR [*object*] Function
CAAADR [*object*] Function
CDAADR [*object*] Function
CADADR [*object*] Function
CDDADR [*object*] Function
CAADDR [*object*] Function
CDADDR [*object*] Function
CADDDR [*object*] Function
CDDDDR [*object*] Function

Functions for all combinations of CAR and CDR up to four compositions are defined in machine language. The name of each of these functions begins with a *C* and ends with an *R*. In between is a sequence of *A*s and *D*s indicating the composition of cars and cdrs that the function performs.

```
(DEFUN CDAR (OBJ)
  (CDR (CAR OBJ)) )

(DEFUN CADDR (OBJ)
  (CAR (CDR (CDR OBJ))) )

$ (SETQ TREE '(                                    ;Initialize TREE
    (((A . B) . (C . D)) . ((E . F) . (G . H))) .
    (((I . J) . (K . L)) . ((M . N) . (O . P)))) )
((((A . B) C . D) (E . F) G . H)
 ((I . J) K . L) (M . N) O . P)

$ (CAR TREE)                                       ;Car of TREE
(((A . B) C . D) (E . F) G . H)

$ (CDR TREE)                                       ;Cdr of TREE
(((I . J) K . L) (M . N) O . P)

$ (CAAR TREE)                                ;Car of car of TREE
((A . B) C . D)

$ (CDAR TREE)                                ;Cdr of car of TREE
((E . F) G . H)

$ (CADR TREE)                                ;Car of cdr of TREE
((I . J) K . L)

$ (CDDR TREE)                                ;Cdr of cdr of TREE
((M . N) O . P)

$ (CAAAR TREE)                         ;Car of car of car of TREE
(A . B)

$ (CADDDR TREE)                  ;Car of cdr of cdr of cdr of TREE
O

$ (CDADAR TREE)                  ;Cdr of car of cdr of car of TREE
F
```

FIRST [list]	Function
SECOND [list]	Function
THIRD [list]	Function
FOURTH [list]	Function
FIFTH [list]	Function
SIXTH [list]	Function
SEVENTH [list]	Function
EIGHTH [list]	Function
NINTH [list]	Function
TENTH [list]	Function
REST [list]	Function

These selectors return any one of the first ten elements of a list. Note that unlike NTH, they are one-based instead of zero-based. REST is included to complement FIRST.

```
(DEFUN SIXTH (LST)
  (NTH 5 LST) )

(DEFUN REST (LST)
  (CDR LST) )

$ (FIRST '(A B C D E))                    ;The first element
A

$ (SECOND '(A B C D E))                   ;The second element
B

$ (THIRD '(A B C D E))                    ;The third element
C

$ (FOURTH '(A B C D E))                   ;The fourth element
D

$ (REST '(A B C D E))                     ;The rest of list
(B C D E)
```

LAST [*list*] Function

(LAST *list*) returns the last top-level cons of *list*. Note that LAST returns the last *cons*, not the last *element*, of *list*. If *list* is an atom, LAST returns NIL. As the final example shows, the last element can be extracted by taking the CAR of (LAST *list*).

```
(DEFUN LAST (LST)
  ((ATOM LST) NIL)
  ((ATOM (CDR LST)) LST)
  (LAST (CDR LST)) )

$ (LAST '(A B C D))                    ;The last of a list
(D)

$ (LAST '(A B C . D))                  ;The last of an S-expr
(C . D)

$ (LAST 'FOO)                          ;The last of a symbol
NIL

$ (LAST 5)                             ;The last of a number
NIL

$ (CAR (LAST '(A B C)))                ;The last element of a list
C
```

NTHCDR [*n*, *list*] Function
NTH [*n*, *list*] Function

If *n* is zero or a positive integer, (NTHCDR *n list*) returns the *n*th cdr of *list*. (NTH *n list*) returns the *n*th element of *list*, where the car of *list* is the *zeroth* element. NTHCDR and NTH return NIL if *n* is neither zero nor a positive integer, or if *list* has *n* or fewer elements.

```
(DEFUN NTHCDR (N LST)
  ((ZEROP N) LST)
  ((AND (INTEGERP N) (PLUSP N))
    ((ATOM LST) NIL)
    (NTHCDR (SUB1 N) (CDR LST)) ) )

(DEFUN NTH (N LST)
  ((ATOM (NTHCDR N LST)) NIL)
  (CAR (NTHCDR N LST)) )

$ (NTHCDR 0 '(A B C D))                    ;No cdrs of list
(A B C D)

$ (NTH 0 '(A B C D))                       ;Zeroth element
A

$ (NTHCDR 2 '(A B C D))                    ;Two cdrs of list
(C D)

$ (NTH 2 '(A B C D))                       ;Second element
C

$ (NTHCDR 4 '(A B C D))                    ;Four cdrs of list
NIL

$ (NTH 4 '(A B C D))                       ;No fourth element
NIL

$ (NTHCDR 2 '(A B . C))                    ;Two cdrs of tree
C
```

LENGTH [*object*] — Function
LIST-LENGTH [*object*] — Library Function

(LENGTH *object*) returns the length of *object* appropriate for its data type. If *object* is NIL or a list, LENGTH returns the number of *top-level* conses (i.e. elements) in *object*. If object is a circular list, LENGTH returns NIL. If *object* is a symbol, LENGTH returns the number of characters in *object*'s print name string. If *object* is a number, LENGTH returns the number of words (a word equals two bytes) required to store *object*'s number vector.

The definition of LIST-LENGTH is *identical* to that of LENGTH and is included in muLISP solely for Common LISP compatibility.

```
(DEFUN LENGTH (OBJ)
  ((NULL OBJ) 0)
  ((SYMBOLP OBJ)
    (the number of characters in OBJ's print name
        string) )
  ((NUMBERP OBJ)
    (CEILING (ABS OBJ) 65536) )
  (ADD1 (LENGTH (CDR OBJ))) )

$ (LENGTH '(A B C D))                    ;Length of list
4

$ (LENGTH NIL)                           ;Length of empty list
0

$ (LENGTH 'MULISP)                       ;Characters in print name
6

$ (LENGTH 100000)                        ;Words to store number
2

$ (LENGTH -13)                           ;Words to store number
1
```

COUNT [*object, list, test*] Function
COUNT-IF [*test, list*] Function

(COUNT *object list test*) returns the number of elements in *list* for which (*test object element*) is nonNIL. If the *test* argument is NIL or not given, COUNT uses an EQL test. (COUNT-IF *test list*) returns the number of elements in *list* for which (*test element*) is nonNIL.

```
(DEFUN COUNT (OBJ LST TEST)
  (IF (NULL TEST) (SETQ TEST 'EQL))
  (count-aux OBJ LST TEST 0) )

(DEFUN count-aux (OBJ LST TEST COUNTER)
  ((ATOM LST) COUNTER)
  ((FUNCALL TEST OBJ (CAR LST))
    (count-aux OBJ (CDR LST) TEST (ADD1 COUNTER)))
  (count-aux OBJ (CDR LST) TEST COUNTER) )

$ (COUNT 'DOG '(CAT DOG COW PIG DOG ANT))
2                                    ;Number of DOGs in list

$ (COUNT 7 '(5 8 12 7.9 3) '<)
3                                  ;Number of numbers > 7 in list

$ (COUNT 'T '(UNPACK 'SUBSTITUTE))
3                                ;Number of T's in "SUBSTITUTE"

$ (COUNT-IF 'EVENP '(3 -6 8 7 0))
3                                  ;Number of even integers in list

$ (COUNT-IF '(LAMBDA (CHAR)
                (MEMBER CHAR '(A E I O U)))
            (UNPACK 'AUTOMOBILE))
6                          ;Number of vowels in "AUTOMOBILE"
```

MISMATCH [*list*$_1$, *list*$_2$, *test*] Function

(MISMATCH *list*$_1$ *list*$_2$ test) compares the elements of *list*$_1$ and *list*$_2$ using the two argument predicate *test* until a mismatch is found (i.e. until *test* returns NIL), or the end of one or both of the lists is reached. If the *test* argument is NIL or not given, MISMATCH uses an EQL test.

If the lists are of equal length and no mismatch is found, MISMATCH returns NIL. If the lists are of unequal length and no mismatch is found, MISMATCH returns the length of the shorter list. Otherwise, MISMATCH returns the zero-based index in the lists of the first pair of elements that fail to match.

```
(DEFUN MISMATCH (LST1 LST2 TEST)
  ((NULL TEST)
    (mismatch-aux LST1 LST2 'EQL 0) )
  (mismatch-aux LST1 LST2 TEST 0) )

(DEFUN mismatch-aux (LST1 LST2 TEST INDEX)
  ((AND (ATOM LST1) (ATOM LST2)) NIL)
  ((OR (ATOM LST1) (ATOM LST2)) INDEX)
  ((NOT (FUNCALL TEST (CAR LST1) (CAR LST2)))
    INDEX )
  (mismatch-aux (CDR LST1) (CDR LST2) TEST
                (ADD1 INDEX)) )

$ (MISMATCH '(A B C D E) '(A B D C E))
2                                    ;Index of first mismatch

$ (MISMATCH '(A B C D E) '(A B C D))
4                                    ;Length of shorter list

$ (MISMATCH '(5 -4 6 9) '(7 -3 9 8) '<)
3                                    ;Index of first greater than

$ (MISMATCH '(5 -4 6 2) '(7 -3 9 8) '<)
NIL                                  ;All less than
```

MEMBER [*object*, *list*, *test*] Function
MEMBER-IF [*test*, *list*] Function

(MEMBER *object list test*) performs a linear search of *list* looking for an element for which (*test object element*) is nonNIL. If the *test* argument is NIL or not given, MEMBER uses an EQL test. (MEMBER-IF *test list*) searches *list* for an element for which (*test element*) is nonNIL. For both functions, if an element satisfying the test is found, the tail of *list* beginning with that element is returned; otherwise NIL is returned.

Note that in some versions of LISP, MEMBER uses an EQUAL test. Of course, this effect can easily be achieved by the call (MEMBER *object list* 'EQUAL).

```
(DEFUN MEMBER (OBJ LST TEST)
  ((ATOM LST) NIL)
  (IF (NULL TEST) (SETQ TEST 'EQL))
  (FUNCALL TEST OBJ (CAR LST)) LST)
  (MEMBER OBJ (CDR LST) TEST) )

$ (MEMBER 'A '(B C D))        ;A is not EQL to a member of list
NIL

$ (MEMBER 'A '(B A D))        ;A is EQL to a member of list
(A D)

$ (MEMBER '(3 4) '((7 8) (3 4) (5 2)))
NIL                           ;(3 4) is not EQL to a member of list

$ (MEMBER '(3 4) '((7 8) (3 4) (5 2)) 'EQUAL)
((3 4) (5 2))                 ;(3 4) is EQUAL to a member of list

$ (MEMBER-IF 'NUMBERP '(A B 3 C (-7)))
(3 C (-7))                          ;There is a number in list

$ (MEMBER-IF 'MINUSP '(A B 3 C (-7)))
NIL                                 ;There is not a negative in list
```

FIND [*object, list, test*] Function
FIND-IF [*test, list*] Function

(FIND *object list test*) performs a linear search of *list* looking for an element for which (*test object element*) is nonNIL. If the *test* argument is NIL or not given, FIND uses an EQL test. (FIND-IF *test list*) searches *list* for an element for which (*test element*) is nonNIL. For both functions, if an element satisfying the test is found, the element is returned; otherwise NIL is returned.

```
(DEFUN FIND (OBJ LST TEST)
  ((ATOM LST) NIL)
  (IF (NULL TEST) (SETQ TEST 'EQL))
  ((FUNCALL TEST OBJ (CAR LST))
    (CAR LST) )
  (FIND OBJ (CDR LST) TEST) )

$ (FIND 'EAT '(CORN WHEAT OATS RICE) 'FINDSTRING)
WHEAT                          ;Find element containing "EAT"

$ (FIND 7 '(5 7 3 8 12 9 3) '<)
8                                        ;Find first element > 7

$ (SETQ WEIGHT '((SUE 120) (TOM 182) (AL 169)))
((SUE 120) (TOM 182) (AL 169))

$ (FIND 'TOM WEIGHT '(LAMBDA (X Y)(EQ X (CAR Y))))
(TOM 182)                                ;Find Tom's weight

$ (FIND-IF '(LAMBDA (Y) (EQ 'TOM (CAR Y))) WEIGHT)
(TOM 182)                                ;Find Tom's weight

$ (FIND-IF 'MINUSP '(4 0 -6 2 -3 7))
-6                                  ;Find first negative number

$ (FIND-IF '(LAMBDA (X) (MINUSP (CADR X)))
           '((X 3.7) (Y 0.5) (Z -2.4)))
(Z -2.4)                          ;Find first negative coordinate
```

POSITION [*object*, *list*, *test*] Function
POSITION-IF [*test*, *list*] Function

(POSITION *object list test*) performs a linear search of *list* looking for an element for which (*test object element*) is nonNIL. If the *test* argument is NIL or not given, POSITION uses an EQL test. (POSITION-IF *test list*) searches *list* for an element for which (*test element*) is nonNIL. For both functions, if an element satisfying the test is found, the zero-based index of the element is returned; otherwise NIL is returned.

```
(DEFUN POSITION (OBJ LST TEST)
  (IF (NULL TEST) (SETQ TEST 'EQL))
  (position-aux OBJ LST TEST 0) )

(DEFUN position-aux (OBJ LST TEST INDEX)
  ((ATOM LST) NIL)
  ((FUNCALL TEST OBJ (CAR LST)) INDEX)
  (position-aux OBJ (CDR LST) TEST (ADD1 INDEX)) )

$ (POSITION 23 '(2 3 5 7 11 13 17 19 23 29 31 37))
8                                ;Number of primes before 23

$ (POSITION '(A B C) '((R S T) (C A B) (A B C))
            'EQUAL)
2                                ;Position of sublist in list

$ (POSITION '(A B C) '((R S T) (C A B) (A B C)))
NIL                      ;Sublist is not EQL to an element of list

$ (POSITION-IF 'PLUSP '(-2.5 0 X 3.7 -5.3))
3                                ;Position of first positive element

$ (POSITION-IF 'EVENP '(2 3 5 7 11 13))
0                                ;Position of first even element
```

ASSOC [*key*, *alist*, *test*] Function
ASSOC-IF [*test*, *alist*] Function

An association list is a list of pairs (i.e. conses) commonly used in LISP to associate a key with an object. ASSOC and ASSOC-IF are selector functions used to access the object associated with a key satisfying a specified test.

(ASSOC *key alist test*) performs a linear search of *alist* looking for a pair for which (*test key* (CAR *pair*)) is nonNIL. If the *test* argument is NIL or not given, ASSOC uses an EQL test. (ASSOC-IF *test alist*) searches *alist* for a pair for which (*test* (CAR *pair*)) is nonNIL. For both functions, if a pair satisfying the test is found, the pair is returned; otherwise NIL is returned.

```
(DEFUN ASSOC (KEY ALST TEST)
  ((ATOM ALST) NIL)
  ((ATOM (CAR ALST))                          ;If an atom,
    (ASSOC KEY (CDR ALST) TEST) )             ;skip it
  (IF (NULL TEST) (SETQ TEST 'EQL))
  ((FUNCALL TEST KEY (CAAR ALST))             ;If match,
    (CAR ALST) )                              ;return it
  (ASSOC KEY (CDR ALST) TEST) )

$ (SETQ CAPITALS '((USA . WASHINGTON)     ;Initialize alist
          (FRANCE . PARIS) (JAPAN . TOKYO)))
((USA . WASHINGTON) (FRANCE . PARIS)
(JAPAN . TOKYO))

$ (ASSOC 'FRANCE CAPITALS)                ;France's capital
(FRANCE . PARIS)                          ;is Paris

$ (ASSOC 'AUSTRALIA CAPITALS)             ;Australia's capital
NIL                                       ;is not in the alist
```

RASSOC [*key, alist, test*] Function
RASSOC-IF [*test, alist*] Function

(RASSOC *key alist test*) performs a linear search of *alist* looking for a pair for which (*test key* (CDR *pair*)) is nonNIL. If the *test* argument is NIL or not given, RASSOC uses an EQL test. (RASSOC-IF *test alist*) searches *alist* for a pair for which (*test* (CDR *pair*)) is nonNIL. For both functions, if a pair satisfying the test is found, the pair is returned; otherwise NIL is returned. RASSOC is short for Reverse ASSOCiate.

```
(DEFUN RASSOC (KEY ALST TEST)
  ((ATOM ALST) NIL)
  ((ATOM (CAR ALST))                          ;If an atom,
    (RASSOC KEY (CDR ALST) TEST) )              ;skip it
  (IF (NULL TEST) (SETQ TEST 'EQL))
  ((FUNCALL TEST KEY (CDAR ALST))             ;If match,
    (CAR ALST) )                                ;return it
  (RASSOC KEY (CDR ALST) TEST) )
```

```
$ (SETQ CAPITALS '((USA . WASHINGTON)     ;Initialize alist
          (FRANCE . PARIS) (JAPAN . TOKYO)))
((USA . WASHINGTON) (FRANCE . PARIS)
(JAPAN . TOKYO))

$ (RASSOC 'PARIS CAPITALS)                  ;Paris is capital
(FRANCE . PARIS)                              ;of France

$ (RASSOC 'CANBERRA CAPITALS)               ;Canberra's country
NIL                                           ;is not in the alist
```

5.2 CONSTRUCTOR FUNCTIONS

muLISP programs can dynamically generate new data structures using the constructor functions. These functions can create binary trees or linked lists that model the data structure of virtually any problem.

CONS [*object*$_1$, *object*$_2$] Function

(CONS *object*$_1$ *object*$_2$) returns a cons whose car cell points to *object*$_1$ and whose cdr cell points to *object*$_2$. If the value of *FREE-LIST* is a cons, CONS modifies and returns that cons and sets *FREE-LIST* to the cdr of *FREE-LIST* (see Section 5.18).

The correct interpretation of (CONS *object*$_1$ *object*$_2$) depends on how *object*$_2$ is regarded. If *object*$_2$ is a list, CONS creates a list whose first element is *object*$_1$ and whose tail is *object*$_2$. If *object*$_2$ is an atom or a binary tree, CONS creates a tree whose left or car branch is *object*$_1$, and whose right or cdr branch is *object*$_2$. Note that CONS creates a new cons rather than altering either *object*$_1$ or *object*$_2$.

```
(DEFUN CONS (OBJ1 OBJ2)
  ((ATOM *FREE-LIST*)
    (return new cons pointing to OBJ1 and OBJ2) )
  (RPLACA *FREE-LIST* OBJ1)
  (RPLACD (PROG1 *FREE-LIST* (POP *FREE-LIST*))
          OBJ2) )

$ (CONS 'A '(B C D))                    ;Cons A onto (B C D)
(A B C D)

$ (CONS 'A 'B)                          ;Create dotted pair
(A . B)
```

LIST [$object_1$, $object_2$, ..., $object_n$] Function
LIST* [$object_1$, $object_2$, ..., $object_n$] Function

LIST $object_1$ $object_2$ ... $object_n$) constructs and returns the list consisting of the elements $object_1$ through $object_n$. If called without arguments, LIST returns NIL. LIST can take any number of arguments.

LIST* $object_1$ $object_2$ ... $object_n$) conses $object_1$ through $object_{n-1}$ onto $object_n$ and returns the resulting object. LIST* is useful for consing several items onto the front of a list. If called with a single argument, LIST* returns that argument.

```
(DEFUN LIST LST
  ((NULL LST) NIL)
  (CONS (CAR LST) (APPLY 'LIST (CDR LST))) )

(DEFUN LIST* LST
  ((NULL LST) NIL)
  ((NULL (CDR LST))
    (CAR LST) )
  (CONS (CAR LST) (APPLY 'LIST* (CDR LST))) )

$ (LIST 'A 'B 'C 'D)                    ;Create 4 element list
(A B C D)

$ (LIST* 'A 'B 'C 'D)                   ;Create 4 leaf tree
(A B C . D)

$ (LIST 'A 'B '(C D))                   ;Create 3 element list
(A B (C D))

$ (LIST* 'A 'B '(C D))                  ;Create 4 element list
(A B C D)

$ (LIST)                                ;Create 0 element list
NIL
```

MAKE-LIST [*n*, *object*, *list*] Function

(MAKE-LIST *n object list*) constructs and returns a list of *n* elements each initialized to *object* nconced to *list*. *n* defaults to 0, *object* and *list* default to NIL.

```
(DEFUN MAKE-LIST (N OBJ LST)
  ((AND (INTEGERP N) (PLUSP N))
    (CONS OBJ (MAKE-LIST (SUB1 N) OBJ LST)) )
  LST )

$ (MAKE-LIST 4)                          ;Make 4 element list
(NIL NIL NIL NIL)

$ (MAKE-LIST 3 '(A B C))                 ;Make 3 element list
((A B C) (A B C) (A B C))

$ (MAKE-LIST 3 5 '(2 3))                 ;Add three 5's to list
(5 5 5 2 3)
```

APPEND [$list_1$, $list_2$, ..., $list_n$] Function

(APPEND $list_1$ $list_2$ ... $list_n$) constructs and returns a list consisting of the elements of the lists $list_1$, $list_2$, through $list_n$. APPEND copies the top-level conses of each of its arguments except the last. If called with a single argument, APPEND simply returns the argument without copying it. Therefore COPY-LIST rather than APPEND must be used to copy a list.

Note that if APPEND and NCONC, defined in the next section, are given the same arguments, they will return EQUAL lists. However, APPEND uses conses to copy all but its last argument, whereas NCONC actually modifies all but its last argument. Compare the following examples with those given for NCONC in the next section.

```
(DEFUN APPEND (LST1 LST2)
   ((ATOM LST1) LST2)
   (CONS (CAR LST1) (APPEND (CDR LST1) LST2)) )

$ (SETQ FOO '(D E F))                          ;Set FOO to list
(D E F)

$ (APPEND '(A B C) FOO '(G H I))               ;Append lists
(A B C D E F G H I)

$ FOO                                          ;FOO unchanged
(D E F)

$ (APPEND '(A B C D) 'E 'F 'G)                 ;Appending atoms
(A B C D . G)
```

COPY-LIST [*list*] Function
COPY-ALIST [*alist*] Function
COPY-TREE [*object*] Function

(COPY-LIST *list*) copies the *top-level* conses of *list*, returning a list that is EQUAL to *list*.

(COPY-ALIST *alist*) copies the *top-level* conses of *alist* and the top-level cons of each element of *alist* that is a cons, returning a list that is EQUAL to *alist*. To prevent unwanted side-effects, COPY-ALIST should be used to copy association lists.

(COPY-TREE *object*) copies *all* conses of *object*, returning an object that is EQUAL to *object*.

```
(DEFUN COPY-LIST (LST)
  ((ATOM LST) LST)
  (CONS (CAR LST) (COPY-LIST (CDR LST))) )

(DEFUN COPY-ALIST (ALST)
  ((ATOM ALST) ALST)
  ((ATOM (CAR ALST))
    (CONS (CAR ALST) (COPY-ALIST (CDR ALST))) )
  (ACONS (CAAR ALST) (CDAR ALST)
         (COPY-ALIST (CDR ALST))) )

(DEFUN COPY-TREE (OBJ)
  ((ATOM OBJ) OBJ)
  (CONS (COPY-TREE (CAR OBJ))
        (COPY-TREE (CDR OBJ))) )

$ (COPY-LIST '(A B (C . D) E))      ;Does not copy (C . D)
(A B (C . D) E)

$ (COPY-ALIST '(A B (C . D) E))     ;Does copy (C . D)
(A B (C . D) E)

$ (COPY-TREE '(A B (C . D) E))      ;Does copy (C . D)
(A B (C . D) E)
```

FIRSTN [*n*, *list*] **Function**

If *n* is a positive integer, (FIRSTN *n list*) copies and returns the first *n* elements of *list*. FIRSTN returns NIL if *n* is not a positive integer. If *list* has *n* or fewer elements, FIRSTN copies and returns the elements of *list*.

```
(DEFUN FIRSTN (N LST)
  ((AND (INTEGERP N) (PLUSP N))
    ((ATOM LST) NIL)
    (CONS (CAR LST)
          (FIRSTN (SUB1 N) (CDR LST))) ) )

$ (FIRSTN 0 '(SUE JOE ANN BOB))
NIL

$ (FIRSTN 2 '(SUE JOE ANN BOB))
(SUE JOE)

$ (FIRSTN 4 '(SUE JOE ANN BOB))
(SUE JOE ANN BOB)

$ (FIRSTN 5 '(SUE JOE ANN BOB))
(SUE JOE ANN BOB)

$ (FIRSTN 2 '(A B . C))
(A B)

$ (FIRSTN 3 '(A B . C))
(A B)
```

SUBLIST [*list*, *n*, *m*] Function

If *n* and *m* are integers and $0 \leq n \leq m$, (SUBLIST *list n m*) copies and returns the *n*th through *m*th elements of *list*, where the car of *list* is the *zeroth* element. If *m* is not an integer or is greater than or equal to the length of *list*, *m* is equivalent to being one less than the length of *list*. If *n* is not an integer greater than or equal to 0 or if $n > m$, SUBLIST returns NIL.

```
(DEFUN SUBLIST (LST N M)
  ((INTEGERP N)
    ((INTEGERP M)
      (FIRSTN (ADD1 (- M N)) (NTHCDR N LST)) )
    (NTHCDR N LST) ) )

$ (SUBLIST '(A B C D E F) 2 4)                ;2nd thru 4th
(C D E)

$ (SUBLIST '(A B C D E F) 2 2)                ;2nd element
(C)

$ (SUBLIST '(A B C D E F) 2 1)                ;No elements
NIL

$ (SUBLIST '(A B C D E F) 2)                  ;Last 4 elements
(C D E F)

$ (SUBLIST '(A B C D E F) 0 3)                ;First 4 elements
(A B C D)
```

LDIFF [*list*, *tail*] Function

If *tail* is a tail (one of the top-level conses) of *list*, (LDIFF *list tail*) copies and returns the elements of *list* up to, but not including, *tail*. If *tail* is not a tail of *list*, LDIFF copies and returns *list*. In neither case is *list* modified or destroyed. LDIFF is short for list difference.

```
(DEFUN LDIFF (LST TAIL)
  ((ATOM LST) LST)
  ((EQ LST TAIL) NIL)
  (CONS (CAR LST) (LDIFF (CDR LST) TAIL)) )

$ (SETQ NUMS '(1 2 3 4 5))                ;Initialize NUMS
(1 2 3 4 5)

$ (SETQ TAIL-NUMS (CDDDR NUMS))           ;Tail of NUMS
(4 5)

$ (LDIFF NUMS TAIL-NUMS)          ;But TAIL-NUMS of NUMS
(1 2 3)

$ (LDIFF NUMS '(4 5))             ;(4 5) is not a tail of NUMS
(1 2 3 4 5)
```

BUTLAST [*list*, *n*] **Function**

If *n* is zero or a positive integer, (BUTLAST *list n*) copies and returns all but the last *n* elements of *list*. If *n* is omitted or is neither zero nor a positive integer, BUTLAST copies and returns all but the last element of *list*. See NBUTLAST for a destructive version of this function.

```
(DEFUN BUTLAST (LST N)
  ((AND (INTEGERP N) (>= N 0))
    (FIRSTN (- (LENGTH LST) N) LST) )
  (BUTLAST LST 1) )

$ (BUTLAST '(A B C D E))                    ;But last element
(A B C D)

$ (BUTLAST '(A B C D E) 2)                  ;But last 2 elements
(A B C)

$ (SETQ FOO '(A B C))                       ;Initialize FOO
(A B C)

$ (BUTLAST FOO)                             ;But last element
(A B)

$ FOO                                       ;FOO unchanged
(A B C)
```

REVERSE [*list*, *object*] Function
REVAPPEND [*list*, *object*] Library Function

(REVERSE *list*) constructs and returns a list consisting of the elements in *list*, but in reverse order. (REVERSE *list object*) returns the reverse of *list* appended to *object*. The result is the same as (APPEND (REVERSE *list*) *object*), but the two argument call on REVERSE is more efficient.

The definition of REVAPPEND is *identical* to that of REVERSE and is included in muLISP solely for Common LISP compatibility.

REVERSE uses consing to construct and return a new list rather than modifying *list*. NREVERSE can also be used to reverse a list, but it destructively modifies the list.

```
(DEFUN REVERSE (LST OBJ)
   ((ATOM LST) OBJ)
   (REVERSE (CDR LST) (CONS (CAR LST) OBJ)) )

$ (REVERSE '(A B C D E))                    ;Reverse list
(E D C B A)

$ (REVERSE '(A B C) '(D E F))               ;Reverse and append
(C B A D E F)

$ (REVERSE '(A B C) 'D)                     ;Reverse and append
(C B A . D)

$ (SETQ FOO '(A B C))                       ;Initialize FOO
(A B C)

$ (REVERSE FOO)                             ;Reverse of FOO
(C B A)

$ FOO                                       ;FOO unchanged
(A B C)
```

REMOVE [*item*, *list*, *test*] Function
REMOVE-IF [*test*, *list*] Function

(REMOVE *item list test*) returns a copy of *list* with all elements for which (*test item element*) is nonNIL removed. If the *test* argument is NIL or not given, REMOVE uses an EQL test. (REMOVE-IF *test list*) returns a copy of *list* with all elements for which (*test element*) is nonNIL removed. REMOVE and REMOVE-IF do not destructively alter *list*. See DELETE and DELETE-IF for destructive versions of these functions.

```
(DEFUN REMOVE (ITEM LST TEST)
  ((ATOM LST) LST)
  (IF (NULL TEST) (SETQ TEST 'EQL))
  ((FUNCALL TEST ITEM (CAR LST))
    (REMOVE ITEM (CDR LST) TEST) )
  (CONS (CAR LST) (REMOVE ITEM (CDR LST) TEST)) )

$ (REMOVE 'RAT '(DOG RAT CAT PIG RAT COW))
(DOG CAT PIG COW)                          ;Removes all RATs

$ (REMOVE '(2 5) '((5 2) (2 5) (2 3)))
((5 2) (2 5) (2 3))                        ;Lists are not EQL

$ (REMOVE '(2 5) '((5 2) (2 5) (2 3)) 'EQUAL)
((5 2) (2 3))                              ;Lists are EQUAL

$ (REMOVE-IF 'MINUSP '(-2 0 7 -0.1 3))
(0 7 3)                                    ;Removes all negatives
```

REMOVE-DUPLICATES [*list*, *test*] Function

(REMOVE-DUPLICATES *list test*) returns a copy of *list* with all duplicate elements removed. Duplicate elements are those for which (*test element*$_1$ *element*$_2$) is nonNIL. If the *test* argument is NIL or not given, REMOVE-DUPLICATES uses an EQL test. See DELETE-DUPLICATES for a destructive version of this function.

```
(DEFUN REMOVE-DUPLICATES (LST TEST)
  ((ATOM LST) LST)
  ((MEMBER (CAR LST) (CDR LST) TEST)
    (REMOVE-DUPLICATES (CDR LST) TEST) )
  (CONS (CAR LST)
        (REMOVE-DUPLICATES (CDR LST) TEST)) )

$ (SETQ NUMS '(4 2 3 4 5 3 0))          ;Initialize NUMS
(4 2 3 4 5 3 0)

$ (REMOVE-DUPLICATES NUMS)              ;Remove duplicates
(2 4 5 3 0)

$ NUMS                                  ;NUMS unchanged
(4 2 3 4 5 3 0)
```

SUBSTITUTE [*new, old, list, test*] Function
SUBSTITUTE-IF [*new, test, list*] Function

(SUBSTITUTE *new old list test*) returns a *top-level* copy of *list*, replacing with *new* the elements of *list* for which (*test old element*) is nonNIL. If the *test* argument is NIL or not given, SUBSTITUTE uses an EQL test. (SUBSTITUTE-IF *new test list*) returns a top-level copy of *list*, replacing with *new* all the elements of *list* for which (*test element*) is nonNIL. SUBSTITUTE and SUBSTITUTE-IF do not destructively alter *list*. See NSUBSTITUTE and NSUBSTITUTE-IF for destructive versions of these functions.

```
(DEFUN SUBSTITUTE (NEW OLD LST TEST)
  ((ATOM LST) LST)
  (IF (NULL TEST) (SETQ TEST 'EQL))
  ((FUNCALL TEST OLD (CAR LST))
    (CONS NEW
          (SUBSTITUTE NEW OLD (CDR LST) TEST)) )
  (CONS (CAR LST)
        (SUBSTITUTE NEW OLD (CDR LST) TEST)) )

$ (SETQ EXPN '(+ 3 X (* 2 X) X))          ;Initialize EXPN
(+ 3 X (* 2 X) X)

$ (SUBSTITUTE 5 'X EXPN)                  ;Replace top-level Xs
(+ 3 5 (* 2 X) 5)

$ EXPN                                    ;EXPN unchanged
(+ 3 X (* 2 X) X)

$ (SUBSTITUTE-IF 0 'MINUSP '(3 -2 (-7 5) -4.7))
(3 0 (-7 5) 0)                    ;Replace top-level negatives

$ (SUBSTITUTE-IF 0                ;Replace small numbers with 0
    '(LAMBDA (N) (AND (NUMBERP N) (< -0.1 N 0.1)))
    '(0.5 -0.09 (1.1 0.03) 0.2))
(0.5 0 (1.1 0.03) 0.2)
```

JBST [*new, old, object, test*] Function
JBST-IF [*new, test, object*] Function

UBST *new old object test*) returns a copy of *object*, replacing with *new* l subexpressions of *object* for which (*test old subexpression*) is nonNIL. the *test* argument is NIL or not given, SUBST uses an EQL test. UBST-IF *new test object*) returns a copy of *object*, replacing with *new* l subexpressions of *object* for which (*test subexpression*) is nonNIL. To revent unnecessary consing, SUBST and SUBST-IF use as much of the iginal object as possible in the result; however in no case is *object* structively altered. See NSUBST and NSUBST-IF for destructive ersions of these functions.

```
DEFUN SUBST (NEW OLD OBJ TEST)
  (IF (NULL TEST) (SETQ TEST 'EQL))
  ((FUNCALL TEST OLD OBJ) NEW)
  ((ATOM OBJ) OBJ)
  (LET ( (LEFT (SUBST NEW OLD (CAR OBJ) TEST))
         (RIGHT (SUBST NEW OLD (CDR OBJ) TEST)) )
    ((AND (EQ LEFT (CAR OBJ))
          (EQ RIGHT (CDR OBJ)))
      OBJ)
    (CONS LEFT RIGHT) ) )

$ (SETQ EXPN '(+ 3 X (* 2 X) X))          ;Initialize EXPN
(+ 3 X (* 2 X) X)

$ (SUBST 5 'X EXPN)                        ;Replace all Xs
(+ 3 5 (* 2 5) 5)

$ EXPN                                     ;EXPN unchanged
(+ 3 X (* 2 X) X)

$ (SUBST-IF 0 'MINUSP '(3 -2 (-7 5) -4.7))
(3 0 (0 5) 0)                              ;Replace all negatives

$ (SUBST-IF 0                              ;Replace small numbers with 0
    '(LAMBDA (N) (AND (NUMBERP N) (< -0.1 N 0.1)))
    '(0.5 -0.09 (1.1 0.03) 0.2))
(0.5 0 (1.1 0) 0.2)
```

SUBLIS [*alist, object, test*] **Function**

(SUBLIS *alist object test*) returns a copy of *object*, replacing all subexpressions of *object* for which there is a key in *alist* such that (*test key subexpression*) is nonNIL with the value associated with the key. If the *test* argument is NIL or not given, SUBLIS uses an EQL test. To prevent unnecessary consing, SUBLIS uses as much of the original object as possible in the result; however in no case is *object* destructively altered. See NSUBLIS for a destructive version of this function.

```
(DEFUN SUBLIS (ALST OBJ TEST)
  ((ASSOC OBJ ALST TEST)
    (CDR (ASSOC OBJ ALST TEST)) )
  ((ATOM OBJ) OBJ)
  (LET ( (LEFT (SUBLIS ALST (CAR OBJ) TEST))
         (RIGHT (SUBLIS ALST (CDR OBJ) TEST)) )
    ((AND (EQ LEFT (CAR OBJ))
          (EQ RIGHT (CDR OBJ)))
      OBJ )
    (CONS LEFT-RIGHT) ) )

$ (SETQ EXPN '(* (+ X Y) (^ X 2) Y))     ;Initialize EXPN
(* (+ X Y) (^ X 2) Y)

$ (SETQ VALS '((X . 5) (Y . 3)))         ;Initialize VALS
((X . 5) (Y . 3))

$ (SUBLIS VALS EXPN)                     ;Substitute for X and Y
(* (+ 5 3) (^ 5 2) 3)

$ EXPN                                   ;EXPN unchanged
(* (+ X Y) (^ X 2) Y)
```

ADJOIN [*object*, *list*, *test*] Function

If *object* is a member of *list* (i.e. there is an *element* of *list* for which (*test object element*) is a nonNIL), (ADJOIN *object list test*) returns *list*. Otherwise, ADJOIN returns (CONS *object list*). If the *test* argument is NIL or not given, ADJOIN uses an EQL test.

```
(DEFFUN ADJOIN (OBJ LST TEST)
  ((MEMBER OBJ LST TEST) LST)
  (CONS OBJ LST) )

$ (ADJOIN 'DOG '(CAT COW DOG PIG))
(CAT COW DOG PIG)

$ (ADJOIN 'DOG '(CAT COW RAT PIG))
(DOG CAT COW RAT PIG)
```

UNION [*list*$_1$, *list*$_2$, *test*] Function
INTERSECTION [*list*$_1$, *list*$_2$, *test*] Function

(UNION *list*$_1$ *list*$_2$ *test*) constructs and returns a list of the elements that are in *list*$_1$ *or list*$_2$. *test* is a two argument predicate used to determine if the elements from each list are the same so both elements will not be repeated in the result. If the *test* argument is NIL or not given, UNION uses an EQL test.

(INTERSECTION *list*$_1$ *list*$_2$ *test*) constructs and returns a list of the elements that are in *list*$_1$ *and list*$_2$. *test* is a two argument predicate used to determine if the elements from each list are the same so one of them can be included the result. If the *test* argument is NIL or not given, INTERSECTION uses an EQL test.

```
(DEFUN UNION (LST1 LST2 TEST)
  ((ATOM LST1) LST2)
  ((MEMBER (CAR LST1) LST2 TEST)
    (UNION (CDR LST1) LST2 TEST) )
  (CONS (CAR LST1) (UNION (CDR LST1) LST2 TEST)) )

(DEFUN INTERSECTION (LST1 LST2 TEST)
  ((ATOM LST1) NIL)
  ((MEMBER (CAR LST1) LST2 TEST)
    (CONS (CAR LST1)
          (INTERSECTION (CDR LST1) LST2 TEST)) )
  (INTERSECTION (CDR LST1) LST2 TEST) )

$ (UNION '(LISP FORTH APL) '(APL PASCAL LISP C))
(FORTH APL PASCAL LISP C)

$ (INTERSECTION '(3 8 2 0) '(5 2 7 3 4))
(3 2)
```

SET-DIFFERENCE [*list*$_1$, *list*$_2$, *test*] Function
SET-EXCLUSIVE-OR [*list*$_1$, *list*$_2$, *test*] Function

(SET-DIFFERENCE *list*$_1$ *list*$_2$ *test*) constructs and returns a list of those elements of *list*$_1$ that do not appear in *list*$_2$. *test* is a two argument predicate used to determine if the elements of *list*$_1$ are in *list*$_2$ so that such elements will *not* be included in the result. If the *test* argument is NIL or not given, SET-DIFFERENCE uses an EQL test.

(SET-EXCLUSIVE-OR *list*$_1$ *list*$_2$ *test*) constructs and returns a list of those elements of *list*$_1$ and *list*$_2$ that do not appear in both lists. *test* is a two argument predicate used to determine if an element exists in both lists so that it will *not* be included in the result. If the *test* argument is NIL or not given, SET-EXCLUSIVE-OR uses an EQL test.

```
(DEFUN SET-DIFFERENCE (LST1 LST2 TEST)
  ((ATOM LST1) NIL)
  ((MEMBER (CAR LST1) LST2 TEST)
    (SET-DIFFERENCE (CDR LST1) LST2 TEST) )
  (CONS (CAR LST1)
        (SET-DIFFERENCE (CDR LST1) LST2 TEST)) )

(DEFUN SET-EXCLUSIVE-OR (LST1 LST2 TEST)
  (NCONC (SET-DIFFERENCE LST1 LST2 TEST)
         (SET-DIFFERENCE LST2 LST1 TEST)) )

$ (SET-DIFFERENCE '(APL PASCAL LISP C)
                  '(LISP FORTH APL))
(PASCAL C)

$ (SET-DIFFERENCE '(4 2 8 5 6) '(3 6 7 2))
(4 8 5)

$ (SET-DIFFERENCE '(3 6 7 2) '(4 2 8 5 6))
(3 7)

$ (SET-EXCLUSIVE-OR '(4 2 8 5 6) '(3 6 7 2))
(4 8 5 3 7)
```

ACONS [*key*, *object*, *alist*] Function
PAIRLIS [*keys*, *objects*, *alist*] Function

(ACONS *key object alist*) constructs the pair (*key* . *object*) and conses it on the head of *alist* and returns the resulting association list. ACONS simplifies the construction of association lists (see the function ASSOC in the previous section).

(PAIRLIS *keys objects alist*) constructs pairs from the elements of *keys* and *objects* and returns the resulting association list appended onto *alist*. If the lists are of unequal length, the construction stops when the end of the shorter list is reached.

```
(DEFUN ACONS (KEY OBJ ALST)
  (CONS (CONS KEY OBJ) ALST) )

(DEFUN PAIRLIS (KEYS OBJS ALST)
  ((OR (ATOM KEYS) (ATOM OBJS)) ALST)
  (ACONS (CAR KEYS) (CAR OBJS)
         (PAIRLIS (CDR KEYS) (CDR OBJS) ALST)) )
```

```
                                          ;Add new pair to alist
$ (ACONS 'EYES 'BLUE '((WEIGHT . 170) (AGE . 39)))
((EYES . BLUE) (WEIGHT . 170) (AGE . 39))

$ (PAIRLIS '(X Y Z) '(5 2 8))             ;Make an alist
((X . 5) (Y . 2) (Z . 8))
```

OBLIST [] Function
STACK-LIST [] Function

(OBLIST) constructs and returns the object list, a list of the currently active symbols in the system. (It would be more appropriate to call it the symbol list but we bow to tradition.) The symbols are listed in the reverse order from which they were read in and/or generated by the string functions: newer symbols to the left of older symbols. Note that unneeded symbols are automatically collected by the garbage collector (see the Memory Management section of Chapter 4).

(STACK-LIST) constructs and returns a list of the objects currently on the muLISP variable stack (see the Memory Management section of Chapter 4). The objects are listed in order: more recently pushed objects to the left of less recently pushed objects. STACK-LIST is primarily useful during an error break to help determine what caused the error.

```
(DEFUN OBLIST ()
  (construct and return a list of all the symbols
      currently in the system) )

(DEFUN STACK-LIST ()
  (construct and return a list of the objects
      pointed to by the variable stack) )

$ (LENGTH (OBLIST))                    ;Right after muLISP startup,
412                                    ;this is number of primitive symbols

$ (COUNT-IF 'GETD (OBLIST))            ;Right after muLISP startup,
357                                    ;this is number of primitive functions
```

5.3 MODIFIER FUNCTIONS

Modifier functions actually redirect pointers in LISP data structures. Thus, modifier functions are used primarily for their *effect* rather than for the *value* they return. By modifying an existing data structure, they can eliminate the need for consing together a whole new structure when only a small part needs to be changed. Since modifier functions can easily produce unwanted side-effects, they should be used only by more experienced LISP programmers.

RPLACA [$object_1$, $object_2$] Function

If $object_1$ is neither the symbol NIL nor a number, (RPLACA $object_1$ $object_2$) replaces the car cell of $object_1$ with a pointer to $object_2$ and returns the modified $object_1$.

The effect of the replacement depends on $object_1$'s type. If $object_1$ is a list, the first element of the list is replaced by $object_2$. If $object_1$ is a binary tree, the left branch of the tree is replaced by $object_2$. If $object_1$ is a nonNIL symbol, the symbol's value cell is set to $object_2$. In all cases, the just modified $object_1$ is returned.

```
(DEFUN RPLACA (OBJ1 OBJ2)
  ((OR (NUMBERP OBJ1) (NULL OBJ1)) OBJ1)
  (replace car cell of OBJ1 with pointer to OBJ2)
  OBJ1 )

$ (SETQ FOO '(A B C))                    ;Initialize FOO
(A B C)

$ (RPLACA FOO 'D)                        ;Replace car of FOO
(D B C)

$ FOO                                    ;Newly modified FOO
(D B C)
```

RPLACD [*object*$_1$, *object*$_2$] **Function**

If *object*$_1$ is neither the symbol NIL nor a number, (RPLACD *object*$_1$ *object*$_2$) replaces the cdr cell of *object*$_1$ with a pointer to *object*$_2$ and returns the modified *object*$_1$.

The effect of the replacement depends on *object*$_1$'s type. If *object*$_1$ is a list, the tail of the list is replaced by *object*$_2$. If *object*$_1$ is a binary tree, the right branch of the tree is replaced by *object*$_2$. If *object*$_1$ is a nonNIL symbol, the symbol's property list is replaced by *object*$_2$. In all cases, the just modified *object*$_1$ is returned.

```
(DEFUN RPLACD (OBJ1 OBJ2)
  ((OR (NUMBERP OBJ1) (NULL OBJ1)) OBJ1)
  (replace cdr cell of OBJ1 with pointer to OBJ2)
  OBJ1 )

$ (SETQ FOO '(A B C))                    ;Initialize FOO
(A B C)

$ (RPLACD FOO '(D E))                    ;Replace cdr of FOO
(A D E)

$ FOO                                    ;Newly modified FOO
(A D E)

$ (RPLACD 'AL '((AGE . 36) (WEIGHT . 169)))
AL                                       ;Replace AL's property list

$ (GET 'AL 'WEIGHT)                      ;Get AL's weight
169
```

FILL [list, object] Function
REPLACE [$list_1$, $list_2$] Function

By modifying the car of the top-level conses of list, (FILL list object) replaces the elements of list with object and returns the modified list.

By modifying the car of the top-level conses of list1, (REPLACE $list_1$ $list_2$) replaces successive elements of $list_1$ with successive elements of $list_2$. If the lists are of unequal lengths, replacement ceases when the end of the shorter list is reached.

```
(DEFUN FILL (LST OBJ)
  ((ATOM LST) LST)
  (FILL (CDR LST) OBJ)
  (RPLACA LST OBJ) )

(DEFUN REPLACE (LST1 LST2)
  ((OR (ATOM LST1) (ATOM LST2)) LST1)
  (REPLACE (CDR LST1) (CDR LST2))
  (RPLACA LST1 (CAR LST2)) )

$ (SETQ FOO '(X Y Z W))                    ;Initialize FOO
(X Y Z W)

$ (REPLACE FOO '(1 2 3))          ;Replace 3 elements of FOO
(1 2 3 W)

$ FOO                                      ;Newly modified FOO
(1 2 3 W)

$ (FILL FOO 4)                             ;Fill FOO with 4's
(4 4 4 4)

$ FOO                                      ;Newly modified FOO
(4 4 4 4)
```

NCONC [$list_1$, $list_2$, ..., $list_n$] Function

NCONC $list_1$ $list_2$ ... $list_n$) returns a list consisting of the elements of the ists $list_1$, $list_2$, through $list_n$ in this order, by modifying the cdr of the ast top-level cons of $list_1$ through $list_{n-1}$.

Note that if NCONC and APPEND, defined in the previous section, are given the same arguments, they will return EQUAL lists. However, APPEND uses conses to copy all but its last argument, whereas NCONC ictually modifies all but its last argument by redirecting their final cdr :ells.

:NCONC *list list*) will turn *list* into a circular list. If an attempt is then nade to display *list*, the display will continue indefinitely or until halted oy a console interrupt (see Chapter 4). Compare the following examples with those given for APPEND in the previous section.

```
(DEFUN NCONC (LST1 LST2)
  ((ATOM LST1) LST2)
  (RPLACD (LAST LST1) LST2)
  LST1 )

$ (SETQ FOO '(D E F))                    ;Initialize FOO
(D E F)

$ (NCONC '(A B C) FOO '(G H I))          ;Concatenate lists
(A B C D E F G H I)

$ FOO                                    ;Newly modified FOO
(D E F G H I)

$ (NCONC '(A B C D) 'E 'F 'G)            ;Nconcing atoms
(A B C D . G)
```

NBUTLAST [*list*, *n*] Function

If *n* is zero or a positive integer, (NBUTLAST *list n*) returns a list consisting of all but the last *n* elements of *list*, by replacing the cdr of the *n*th cons from the end of *list* with NIL. *n* defaults to 1. See BUTLAST for a non-destructive version of this function.

```
(DEFUN NBUTLAST (LST N)
  ((AND (INTEGERP N) (>= N 0))
    (NREVERSE (NTHCDR N (NREVERSE LST))) )
  (NBUTLAST LST 1) )

$ (NBUTLAST '(A B C D))                    ;Remove last element
(A B C)

$ (NBUTLAST '(A B C D) 2)                  ;Remove last 2 elements
(A B)

$ (SETQ FOO '(A B C))                      ;Initialize FOO
(A B C)

$ (NBUTLAST FOO)                           ;Remove last element
(A B)

$ FOO                                      ;Newly modified FOO
(A B)

$ (NBUTLAST FOO 2)                         ;Remove last 2 elements
NIL

$ FOO                                      ;FOO can not be made NIL!
(A B)                                      ;Should have used a SETQ
```

NREVERSE [*list*, *object*] — Function
NRECONC [*list*, *object*] — Library Function

By modifying the top-level conses of *list*, (NREVERSE *list*) reverses the elements in *list*. (NREVERSE *list object*) returns the nreverse of *list* concatenated to *object*. The result is the same as (NCONC (NREVERSE *list*) *object*), but the two argument call on NREVERSE is more efficient.

The definition of NRECONC is *identical* to that of NREVERSE and is included in muLISP solely for Common LISP compatibility.

NREVERSE modifies the top-level conses of *list* and uses no additional conses. REVERSE can also be used to reverse a list, but it uses consing to construct a new list.

```
(DEFUN NREVERSE (LST OBJ)
  ((ATOM LST) OBJ)
  (NREVERSE (CDR LST) (RPLACD LST OBJ)) )

$ (NREVERSE '(A B C D E))                    ;Reverse list
(E D C B A)

$ (NREVERSE '(A B C) '(D E F))               ;Reverse and nconc
(C B A D E F)

$ (NREVERSE '(A B C) 'D)                     ;Reverse and nconc
(C B A . D)

$ (SETQ FOO '(A B C))                        ;Initialize FOO
(A B C)

$ (NREVERSE FOO)                             ;Reverse FOO
(C B A)

$ FOO                                        ;Newly modified FOO
(A)                                          ;is not reversed list!
```

DELETE [*item*, *list*, *test*] Function
DELETE-IF [*test*, *list*] Function

By modifying the cdr of the top-level conses of *list*, (DELETE *item list test*) deletes all elements of *list* for which (*test item element*) is nonNIL. If the *test* argument is NIL or not given, DELETE uses an EQL test. By modifying the cdr of the top-level conses of *list*, (DELETE-IF *test list*) deletes all elements of *list* for which (*test element*) is nonNIL. See REMOVE and REMOVE-IF for non-destructive versions of these functions.

```
(DEFUN DELETE (ITEM LST TEST)
  (IF (NULL TEST) (SETQ TEST 'EQL))
  ((ATOM LST) LST)
  ((FUNCALL TEST ITEM (CAR LST))
    (DELETE ITEM (CDR LST) TEST) )
  (delete-aux ITEM LST LST TEST) )

(DEFUN delete-aux (ITEM LST TAIL TEST)
  ((ATOM (CDR TAIL)) LST)
  ((FUNCALL TEST ITEM (CADR TAIL))
    (RPLACD TAIL (CDDR TAIL))
    (delete-aux ITEM LST TAIL TEST) )
  (delete-aux ITEM LST (CDR TAIL) TEST) )

$ (DELETE 'RAT '(DOG RAT CAT PIG RAT COW))
(DOG CAT PIG COW)                          ;Deletes all RATs

$ (DELETE '(2 5) '((5 2) (2 5) (2 3)))
((5 2) (2 5) (2 3))                        ;Lists are not EQL

$ (DELETE '(2 5) '((5 2) (2 5) (2 3)) 'EQUAL)
((5 2) (2 3))                              ;Lists are EQUAL

$ (DELETE-IF 'MINUSP '(-2 0 7 -0.1 3))
(0 7 3)                                    ;Deletes all negatives
```

REPLACE-NTH [*item*, *list*, *n*] Function
INSERT-NTH [*item*, *list*, *n*] Function
DELETE-NTH [*list*, *n*] Function

By modifying *list*, (REPLACE-NTH *item list n*) replaces the *n*th element of *list* with *item*, where the car of *list* is the 0th element of *list*. If *list* has *n* or fewer elements, REPLACE-NTH returns *list*.

By modifying *list*, (INSERT-NTH *item list n*) inserts *item* before the *n*th element of *list*. If *list* has fewer than *n* elements, INSERT-NTH returns *list*.

By modifying *list*, (DELETE-NTH *list n*) deletes the *n*th element of *list*. If *list* has *n* or fewer elements, DELETE-NTH returns *list*.

```
(DEFUN INSERT-NTH (ITEM LST N)
  ((AND (INTEGERP N) (>= N 0))
    ((ZEROP N)  (CONS ITEM LST) )
    ((ATOM LST) LST)
    (RPLACD LST
         (INSERT-NTH ITEM (CDR LST) (SUB1 N))) ) )

(DEFUN DELETE-NTH (LST N)
  ((AND (INTEGERP N) (>= N 0))
    ((ATOM LST) LST)
    ((ZEROP N)  (CDR LST) )
    (RPLACD LST
            (DELETE-NTH (CDR LST) (SUB1 N))) ) )

$ (REPLACE-NTH 'A '(B C D E) 2)      ;Replace 2nd element
(B C A E)

$ (INSERT-NTH 'A '(B C D E) 2)       ;Insert before 2nd
(B C A D E)

$ (DELETE-NTH '(A B C D) 1)          ;Delete 1st element
(A C D)

$ (DELETE-NTH '(A B C D) 4)          ;Delete 4th element
(A B C D)
```

DELETE-DUPLICATES [*list*, *test*] **Function**

By rearranging the cdr of the top-level conses of *list*, (DELETE-DUPLICATES *list test*) removes duplicate elements of *list*. Duplicate elements are those for which (*test element*$_1$ *element*$_2$) is nonNIL. If the *test* argument is NIL or not given, DELETE-DUPLICATES uses an EQL test. See REMOVE-DUPLICATES for a non-destructive version of this function.

```
(DEFUN DELETE-DUPLICATES (LST TEST)
  ((ATOM LST) LST)
  ((MEMBER (CAR LST) (CDR LST) TEST)
    (DELETE-DUPLICATES (CDR LST) TEST) )
  (delete-duplicates-aux LST LST TEST) )

(DEFUN delete-duplicates-aux (LST TAIL TEST)
  ((ATOM (CDR TAIL)) LST)
  ((MEMBER (CADR TAIL) (CDDR TAIL) TEST)
    (RPLACD TAIL (CDDR TAIL))
    (delete-duplicates-aux LST TAIL TEST) )
  (delete-duplicates-aux LST (CDR TAIL) TEST) )

$ (SETQ NUMS '(4 2 3 4 5 3 0))          ;Initialize NUMS
(4 2 3 4 5 3 0)

$ (DELETE-DUPLICATES NUMS)              ;Delete duplicates
(2 4 5 3 0)

$ NUMS                                  ;Newly modified NUMS
(4 2 4 5 3 0)                           ;Should have used a SETQ

$ (SETQ NUMS (DELETE-DUPLICATES NUMS))  ;Try again
(2 4 5 3 0)

$ NUMS                                  ;Newly modified NUMS
(2 4 5 3 0)
```

NSUBSTITUTE [*new*, *old*, *list*, *test*] Function
NSUBSTITUTE-IF [*new*, *test*, *list*] Function

By modifying the car of the top-level conses of *list*, (NSUBSTITUTE *new old list test*) replaces with *new* the elements of *list* for which (*test old element*) is nonNIL. If the *test* argument is NIL or not given, NSUBSTITUTE uses an EQL test. By modifying the top-level conses of *list*, (NSUBSTITUTE-IF *new test list*) replaces with *new* the elements of *list* for which (*test element*) is nonNIL. See SUBSTITUTE and SUBSTITUTE-IF for non-destructive versions of these functions.

```
(DEFUN NSUBSTITUTE (NEW OLD LST TEST)
  ((ATOM LST) LST)
  (IF (NULL TEST) (SETQ TEST 'EQL))
  ((FUNCALL TEST OLD (CAR LST))
    (NSUBSTITUTE NEW OLD (CDR LST) TEST)
    (RPLACA LST NEW) )
  (NSUBSTITUTE NEW OLD (CDR LST) TEST)
  LST )

$ (NSUBSTITUTE 'CANNIBALS 'NOUN          ;Replace NOUNs
                '(NOUN LIKE TO EAT NOUN))
(CANNIBALS LIKE TO EAT CANNIBALS)

$ (SETQ EXPN '(+ 3 X (* 2 X) X))         ;Initialize EXPN
(+ 3 X (* 2 X) X)

$ (NSUBSTITUTE 5 'X EXPN)                ;Replace top-level Xs
(+ 3 5 (* 2 X) 5)

$ EXPN                                   ;Newly modified EXPN
(+ 3 5 (* 2 X) 5)

                                ;Replace small magnitude numbers with 0
$ (NSUBSTITUTE-IF 0
    '(LAMBDA (N) (AND (NUMBERP N) (< -0.1 N 0.1)))
    '(0.5 -0.09 (1.1 0.03) 0.2))
(0.5 0 (1.1 0.03) 0.2)
```

NSUBST [*new*, *old*, *object*, *test*] Function
NSUBST-IF [*new*, *test*, *object*] Function

By modifying the conses of *object*, (NSUBST *new old object test*) replaces with *new* all subexpressions of *object* for which (*test old subexpression*) is nonNIL. If the *test* argument is NIL or not given, NSUBST uses an EQL test. By modifying the conses of *object*, (NSUBST-IF *new test object*) replaces with *new* all subexpressions of *object* for which (*test subexpression*) is nonNIL. See SUBST and SUBST-IF for non-destructive versions of these functions.

```
(DEFUN NSUBST (NEW OLD OBJ TEST)
  (IF (NULL TEST) (SETQ TEST 'EQL))
  ((FUNCALL TEST OLD OBJ) NEW)
  ((ATOM OBJ) OBJ)
  (RPLACA OBJ (NSUBST NEW OLD (CAR OBJ) TEST))
  (RPLACD OBJ (NSUBST NEW OLD (CDR OBJ) TEST)) )

$ (SETQ EXPN '(+ 3 X (* 2 X) X))          ;Initialize EXPN
(+ 3 X (* 2 X) X)

$ (NSUBST 5 'X EXPN)                      ;Replace all Xs
(+ 3 5 (* 2 5) 5)

$ EXPN                                    ;Newly modified EXPN
(+ 3 5 (* 2 5) 5)

                          ;Replace small magnitude numbers with 0
$ (NSUBST-IF 0
    '(LAMBDA (N) (AND (NUMBERP N) (< -0.1 N 0.1)))
    '(0.5 -0.09 (1.1 0.03) 0.2))
(0.5 0 (1.1 0) 0.2)
```

NSUBLIS [*alist, object, test*] **Function**

By modifying the conses of object, (NSUBLIS *alist object test*) replaces all subexpressions of *object* for which there is a key in *alist* such that (*test key subexpression*) is nonNIL with the value associated with the key. If the *test* argument is NIL or not given, NSUBLIS uses an EQL test. See SUBLIS for a non-destructive version of this function.

```
(DEFUN NSUBLIS (ALST OBJ TEST)
  ((ASSOC OBJ ALST TEST)
    (CDR (ASSOC OBJ ALST TEST)) )
  ((ATOM OBJ) OBJ)
  (RPLACA OBJ (NSUBLIS ALST (CAR OBJ) TEST))
  (RPLACD OBJ (NSUBLIS ALST (CDR OBJ) TEST)) )

$ (SETQ EXPN '(* (+ X Y) (^ X 2) Y))    ;Initialize EXPN
(* (+ X Y) (^ X 2) Y)

$ (SETQ VALS '((X . 5) (Y . 3)))        ;Initialize VALS
((X . 5) (Y . 3))

$ (NSUBLIS VALS EXPN)                   ;Substitute for X and Y
(* (+ 5 3) (^ 5 2) 3)

$ EXPN                                  ;Newly modified EXPN
(* (+ 5 3) (^ 5 2) 3)
```

TCONC [*dotted-pair*, *object*] **Function**

(TCONC *dotted-pair object*) adds *object* to the end of the list pointed to by the car of *dotted-pair* by modifying the list. TCONC returns *dotted-pair* with the cdr cell modified to point to the new end of the list.

TCONC efficiently adds elements to the tail end of a list, using RPLACD to modify the list. Its first argument is a cons, the car of which points to the beginning of a list and the cdr of which points to the end (i.e. the last cons) of the list. TCONC's second argument is the item to be added to the end of the list. As shown in the examples, the first time TCONC is called, its first argument should be the atom NIL or the list (NIL).

```
(DEFUN TCONC (PAIR OBJ)
  (SETQ OBJ (LIST OBJ))
  ((ATOM PAIR)
    (CONS OBJ OBJ) )
  ((ATOM (CDR PAIR))
    (RPLACA PAIR OBJ)
    (RPLACD PAIR OBJ) )
  (RPLACD (CDR PAIR) OBJ)
  (RPLACD PAIR OBJ) )

$ (SETQ FOO (TCONC NIL 'A))          ;Add A to end of list
((A) A)                              ;Note: SETQ is required

$ (TCONC FOO 'B)                     ;Add B to end of list
((A B) B)

$ (TCONC FOO 'C)                     ;Add C to end of list
((A B C) C)

$ (CAR FOO)                          ;Extract list
(A B C)
```

LCONC [*dotted-pair, list*] Function

(LCONC *dotted-pair list*) adds *list* to the end of the list pointed to by the car of *dotted-pair* by modifying the list. LCONC returns *dotted-pair* with the cdr cell modified to point to the new end of the list.

LCONC efficiently adds lists to the tail end of a list, using RPLACD to modify the list. Its first argument is a cons, the car of which points to the beginning of a list and the cdr of which points to the end (i.e. the last cons) of the list. LCONC's second argument is the item to be added to the end of the list. As shown in the examples, the first time LCONC is called, its first argument should be the atom NIL or the list (NIL).

```
(DEFUN LCONC (PAIR LST)
  ((ATOM LST) PAIR)
  ((ATOM PAIR)
    (CONS LST (LAST LST)) )
  ((ATOM (CDR PAIR))
    (RPLACA PAIR LST)
    (RPLACD PAIR (LAST LST)) )
  (RPLACD (CDR PAIR) LST)
  (RPLACD PAIR (LAST LST)) )

$ (SETQ FOO (LIST NIL))                ;Initialize FOO to (NIL)
(NIL)

$ (LCONC FOO '(A B))                   ;Add (A B) to end of list
((A B) B)                              ;Note: SETQ is not required

$ (LCONC FOO '(C D))                   ;Add (C D) to end of list
((A B C D) D)

$ (LCONC FOO '(E F))                   ;Add (E F) to end of list
((A B C D E F) F)

$ (CAR FOO)                                        ;Extract list
(A B C D E F)
```

SPLIT [*list*] Function

(SPLIT *list*) splits *list* into two lists by replacing the cdr of the middle top-level cons of *list* with NIL. SPLIT returns the tail of *list* beginning where the split was made.

If *list* has an even number of elements, the head and tail of the split will have the same number of elements. If *list* has an odd number of elements, the head will have one more element than the tail.

```
(DEFUN SPLIT (LST)
  ((ATOM LST) LST)
  (split-aux LST (CDR LST)) )

(DEFUN split-aux (HEAD TAIL)
  ((ATOM TAIL)
    (PROG1 (CDR HEAD) (RPLACD HEAD NIL)) )
  (POP TAIL)
  ((ATOM TAIL)
    (PROG1 (CDR HEAD) (RPLACD HEAD NIL)) )
  (split-aux (CDR HEAD) (CDR TAIL)) )

$ (SETQ NAMES '(TOM SUE JOE PAT SAM))      ;Set NAMES
(TOM SUE JOE PAT SAM)

$ (SPLIT NAMES)                            ;SPLIT returns last half
(PAT SAM)

$ NAMES                                    ;NAMES is now first half
(TOM SUE JOE)
```

MERGE [$list_1$, $list_2$, *test*] Function

If *test* is a two argument ordering predicate, (MERGE $list_1$ $list_2$ *test*) destructively merges the elements of $list_1$ and $list_2$ based on *test* and returns the result. (*test* $object_1$ $object_2$) should return NIL if and only if $object_1$ is ordered after $object_2$ or is equal to $object_2$.

The merge is stable in the sense that if an element of $list_1$ is equal to an element of $list_2$, the element of $list_1$ will precede that of $list_2$ in the result.

Since MERGE modifies the top-level conses of the lists, it does not require any additional conses. If the lists should not be destroyed, call MERGE with copies of the lists made by the function COPY-LIST.

```
(DEFUN MERGE (LST1 LST2 TEST)
  ((ATOM LST1) LST2)
  ((ATOM LST2) LST1)
  ((FUNCALL TEST (CAR LST2) (CAR LST1))
    (RPLACD LST2 (MERGE LST1 (CDR LST2) TEST)) )
  (RPLACD LST1 (MERGE (CDR LST1) LST2 TEST)) )
```

```
                                        ;Merge lists of integers
$ (MERGE '(2 4 6 8 10) '(2 3 5 7 11) '<)
(2 2 3 4 5 6 7 8 10 11)
```

```
                                        ;Merge lists of symbols
$ (MERGE '(APPLE ORANGE PEAR)
         '(BANANA GRAPE PLUM)
         'STRING<)
(APPLE BANANA GRAPE ORANGE PEAR PLUM)
```

SORT [*list*, *test*] — Function
STABLE-SORT [*list*, *test*] — Library Function

If *test* is a two argument ordering predicate, (SORT *list test*) destructively sorts the elements of *list* based on *test* and returns the result. (*test* $object_1$ $object_2$) should return NIL if and only if $object_1$ is ordered after $object_2$ or is equal to $object_2$.

SORT performs a list merge sort requiring time proportional to n log n where n is the length of *list* (see Knuth, The Art of Computer Programming, Volume 3, Addison-Wesley, 1973). The sort is also stable in the sense that elements of *list* which *test* finds equal to each other will stay in the same order in the result.

Since SORT modifies the top-level conses of *list*, it does not require any additional conses. If *list* should not be destroyed, call SORT with a copy of *list* made by COPY-LIST.

The definition of STABLE-SORT is *identical* to that of SORT and is included in muLISP solely for Common LISP compatibility.

```
(DEFUN SORT (LST TEST)
  ((NULL LST) NIL)
  ((NULL (CDR LST)) LST)
  (SETQ LST (CONS (SORT (SPLIT LST) TEST)
                  (SORT LST TEST)))
  (MERGE (CDR LST) (CAR LST) TEST) ) .
```

```
                                                ;Sort list of numbers
$ (SORT '(5 2 3/4 7 0 3 -4) '<)
(-4 0 0.75 2 3 5 7)
```

```
                                                ;Sort list of names
$ (SORT '(SMITH JONES MILLER COX SCHMIDT)
        'STRING<)
(COX JONES MILLER SCHMIDT SMITH)
```

5.4 RECOGNIZER FUNCTIONS

Recognizer functions are predicates used to recognize or identify muLISP data objects. They all take a single argument and return a value of either T or NIL.

SYMBOLP [*object*] Function
STRINGP [*object*] Library Function

If *object* is a symbol, (SYMBOLP *object*) returns T, otherwise it returns NIL.

Since no distinction is made between strings and symbols in muLISP, the definition of STRINGP is *identical* to that of SYMBOLP and is included solely for Common LISP compatibility.

```
(DEFUN SYMBOLP (OBJ)
  ((if OBJ is a symbol) T)
  NIL )

$ (SYMBOLP 'DOG)                          ;DOG is a symbol
T

$ (SYMBOLP ())              ;The empty list (= NIL) is a symbol
T

$ (SYMBOLP 100)                        ;Integers are not symbols
NIL

$ (SYMBOLP 2/3)                        ;Numbers are not symbols
NIL

$ (SYMBOLP 4.235)                      ;Numbers are not symbols
NIL

$ (SYMBOLP '(A B C))          ;Nonempty lists are not symbols
NIL
```

INTEGERP [*object*] Function
NUMBERP [*object*] Function
RATIONALP [*object*]' Library Function

If *object* is an integer, (INTEGERP *object*) returns T, otherwise it returns NIL. If *object* is a number, either an integer or a rational, (NUMBERP *object*) returns T, otherwise it returns NIL.

Since rational numbers also serve as floating point numbers in muLISP, the definition of RATIONALP is *identical* to that of NUMBERP and is included solely for Common LISP compatibility.

```
(DEFUN INTEGERP (OBJ)
  ((if OBJ is an integer) T)
  NIL )

(DEFUN NUMBERP (OBJ)
  ((INTEGERP OBJ))
  ((if OBJ is a rational number) T)
  NIL )

$ (INTEGERP 100)                          ;100 is an integer
T

$ (INTEGERP 2/3)                          ;Ratios are not integers
NIL

$ (INTEGERP 4.235)                        ;Decimal numbers are not integers
NIL

$ (NUMBERP 100)                           ;Integers are numbers
T

$ (NUMBERP 2/3)                           ;Ratios are numbers
T

$ (NUMBERP 4.235)                         ;Decimal numbers are numbers
T
```

ATOM [*object*] Function

If *object* is an atom, either a symbol or a number, (ATOM *object*) returns T, otherwise it returns NIL.

```
(DEFUN ATOM (OBJ)
  (OR (SYMBOLP OBJ) (NUMBERP OBJ)) )

$ (ATOM 'DOG)                          ;Symbols are atoms
T

$ (ATOM 100)                           ;Integers are atoms
T

$ (ATOM 2/3)                           ;Numbers are atoms
T

$ (ATOM 4.235)                         ;Numbers are atoms
T

$ (ATOM '(A B C))                      ;Nonempty lists are not atoms
NIL

$ (ATOM '(A . B))                      ;Conses are not atoms
NIL

$ (ATOM ())                            ;The empty list (= NIL ) is an atom
T
```

NULL [*object*] **Function**

If *object* is the empty list, (NULL *object*) returns T, otherwise it returns NIL. The empty or null list is represented by the symbol NIL.

Since the atom NIL represents both the empty list and the truth value false, the functions NULL and NOT always return identical values. However for program readability, it is better to use NULL when testing for the empty list and to use NOT when testing for the truth value false.

```
(DEFUN NULL (OBJ)
  (EQ OBJ NIL) )

$ (NULL NIL)                          ;NIL is the empty list
T

$ (NULL ())                           ;() is the empty list
T

$ (NULL 'DOG)              ;A nonNIL symbol is not the empty list
NIL

$ (NULL '(A B C))          ;A nonempty list is not the empty list
NIL

$ (NULL '(NIL))            ;A nonempty list is not the empty list
NIL
```

CONSP [*object*] Function
LISTP [*object*] Function

If *object* is a cons, (CONSP *object*) returns T, otherwise it returns NIL. If *object* is a list, either a cons or the atom NIL, (LISTP *object*) returns T, otherwise it returns NIL.

```
(DEFUN CONSP (OBJ)
  (NOT (ATOM OBJ)) )

(DEFUN LISTP (OBJ)
  (OR (CONSP OBJ) (NULL OBJ)) )

$ (CONSP 'DOG)                          ;Symbols are not conses
NIL

$ (CONSP 4.235)                         ;Numbers are not conses
NIL

$ (CONSP '(A B C))                      ;Lists are conses
T

$ (CONSP ())                ;The empty list (= NIL) is not a cons
NIL

  (LISTP 'DOG)                          ;Symbols are not lists
NIL

  (LISTP 4.235)                         ;Numbers are not lists
NIL

  (LISTP '(A B C))                      ;(A B C) is a list
T

  (LISTP ())                  ;The empty list (= NIL) is a list
T
```

ENDP [*object*] Function

If *object* is the empty list, (ENDP *object*) returns T. If *object* is a cons, ENDP returns NIL. Otherwise, (i.e. if *object* is an atom), ENDP generates a "Non-list Object" error break. The empty or null list is represented by the symbol NIL.

If ENDP is used instead of NULL to test for the end of lists, the user will be notified if a non-list is ever encountered.

```
(DEFUN ENDP (OBJ)
  ((NULL OBJ))
  ((ATOM OBJ)
    (BREAK (LIST 'ENDP OBJ) "Non-list Object") ) )

$ (ENDP NIL)                          ;NIL is the end of a list
T

$ (ENDP '(A B C))                     ;A nonempty list is not the end
NIL

$ (ENDP 'DOG)                         ;A nonNIL symbol is not a list
Non-list Object Break: (ENDP DOG)
```

5.5 COMPARATOR FUNCTIONS

Comparator functions are predicates used to compare two LISP data objects. They all require two arguments and return a value of either T or NIL.

EQ [$object_1$, $object_2$] Function
NEQ [$object_1$, object$_2$] Function

If $object_1$ is identical to $object_2$, (EQ $object_1$ $object_2$) returns T, otherwise it returns NIL. (NEQ $object_1$ $object_2$) is equivalent to (NOT (EQ $object_1$ $object_2$)).

EQ tests for the identity of its two arguments (i.e. they point to the same data object in memory). Since symbols and small integers (i.e. integers less than 65536 in absolute value) are stored uniquely, EQ is an efficient way to determine if two symbols or two small integers are equal.

```
(DEFUN EQ (OBJ1 OBJ2)
  (= (LOCATION OBJ1) (LOCATION OBJ2)) )

$ (EQ 'Alice 'Alice)                    ;Identical symbols are EQ
T

$ (EQ 7 (+ 3 4))                        ;Equal small integers are EQ
T

$ (EQ 100000 100000)                ;Equal big integers may not be EQ
NIL

$ (EQ 4.75 4.75)                ;Equal decimal numbers may not be EQ
NIL

$ (EQ '(A B C) '(A B C))                ;Equal lists may not be EQ
NIL
```

EQL [*object*$_1$, *object*$_2$] Function
NEQL [*object*$_1$, *object*$_2$] Function

If *object*$_1$ is identical to *object*$_2$ or if both are numbers having the same numeric value, (EQL *object*$_1$ *object*$_2$) returns T, otherwise it returns NIL. (NEQL *object*$_1$ *object*$_2$) is equivalent to (NOT (EQL *object*$_1$ *object*$_2$)).

Since big integers (i.e. integers greater than 65535 in absolute value) and ratios are not stored uniquely, EQL rather EQ should be used to determine if two numbers are equal. Nonatomic arguments are EQL if and only if they point to the same cons in memory.

```
(DEFUN EQL (OBJ1 OBJ2)
  ((AND (NUMBERP OBJ1) (NUMBERP OBJ2))
    (= OBJ1 OBJ2) )
  (= (LOCATION OBJ1) (LOCATION OBJ2)) )

$ (EQL 'Alice 'Alice)              ;Identical symbols are EQL
T

$ (EQL 7 (+ 3 4))                  ;Equal numbers are EQL
T

$ (EQL 100000 100000)              ;Equal numbers are EQL
T

$ (EQL 4.75 4.75)                  ;Equal numbers are EQL
T

$ (EQL '(A B C) '(A B C))          ;Equal lists may not be EQL
NIL
```

EQUAL [*object*$_1$, *object*$_2$, *test*] Function
TREE-EQUAL [*object*$_1$, *object*$_2$, *test*] Library Function

If *object*$_1$ and *object*$_2$ are isomorphic tree structures with leaves such that (*test leave*$_1$ *leave*$_2$) is nonNIL, (EQUAL *object*$_1$ *object*$_2$ *test*) returns T, otherwise it returns NIL. If the *test* argument is NIL or not given, EQUAL uses EQL for *test*. The definition of TREE-EQUAL is *identical* to that of EQUAL and is included in muLISP solely for Common LISP compatibility.

Thus, EQUAL tests for the equality of its two arguments. Two nonatomic arguments are EQUAL if the car branches are EQUAL and the cdr branches are EQUAL. Two EQUAL data structures look the same when printed. Since EQL is faster, it should be used in place of EQUAL in cases where at least one of the arguments is known to be atomic.

```
(DEFUN EQUAL (OBJ1 OBJ2 TEST)
  (IF (NULL TEST) (SETQ TEST 'EQL))
  ((ATOM OBJ1)
    (FUNCALL TEST OBJ1 OBJ2) )
  ((ATOM OBJ2) NIL)
  ((EQUAL (CAR OBJ1) (CAR OBJ2) TEST)
    (EQUAL (CDR OBJ1) (CDR OBJ2) TEST) ) )

$ (EQUAL 'Alice 'Alice)          ;Identical symbols are EQUAL
T

$ (EQUAL 7 (+ 3 4))              ;Equal numbers are EQUAL
T

$ (EQUAL 4.75 4.75)              ;Equal numbers are EQUAL
T

$ (EQUAL '(A B C) '(A B C))      ;Equal lists are EQUAL
T

$ (EQUAL '(A B C) '(C B A))      ;Nonequal lists are not EQUAL
NIL
```

TAILP [*list*$_1$, *list*$_2$] Function
SUBSETP [*list*$_1$, *list*$_2$, *test*] Function

If *list*$_1$ is a sublist (i.e. one of the top-level conses) of *list*$_2$, (TAILP *list*$_1$ *list*$_2$) returns T, otherwise it returns NIL.

If every element of *list*$_1$ is in *list*$_2$, (SUBSETP *list*$_1$ *list*$_2$ *test*) returns T, otherwise it returns NIL. *test* is a two argument predicate used to determine if an element in *list*$_1$ matches an element in *list*$_2$. If the *test* argument is NIL or not given, SUBSETP uses an EQL test.

```
(DEFUN TAILP (LST1 LST2)
  ((EQ LST1 LST2))
  ((ATOM LST2) NIL)
  (TAILP LST1 (CDR LST2)) )

(DEFUN SUBSETP (LST1 LST2 TEST)
  ((ATOM LST1))
  ((MEMBER (CAR LST1) LST2 TEST)
    (SUBSETP (CDR LST1) LST2 TEST) ) )

$ (SETQ FOO '(A B C D))                    ;Initialize FOO
(A B C D)

$ (TAILP '(C D) FOO)                       ;Not a tail of FOO
NIL

$ (SETQ BAZ (CDDR FOO))                    ;Initialize BAZ
(C D)

$ (TAILP BAZ FOO)                          ;BAZ is a tail of FOO
T

$ (SUBSETP '(A B C) '(D A E C B))          ;(A B C) is a subset
T

$ (SUBSETP '(A B C) '(D A E F B))          ;(A B C) is not a subset
NIL
```

ORDERP [$object_1$, $object_2$] **Function**

If $object_1$ is ordered before $object_2$, (ORDERP $object_1$ $object_2$) returns T, otherwise it returns NIL.

ORDERP provides an efficient chronological ordering for atoms that is useful for writing sorting and searching routines. Numbers are ordered before symbols, which are ordered before conses. Two numbers are ordered in increasing numerical order. Two symbols are ordered by comparing their physical memory address (i.e. according to their order of introduction). Two conses are ordered by comparing their physical memory address.

```
(DEFUN ORDERP (OBJ1 OBJ2)
  ((NUMBERP OBJ1)
    ((NUMBERP OBJ2)
      (< OBJ1 OBJ2) )
    T )
  ((NUMBERP OBJ2) NIL)
  (< (LOCATION OBJ1) (LOCATION OBJ2)) )

$ (ORDERP 5 9)                    ;Numbers are ordered numerically
T

$ (ORDERP 'DOG 5)                 ;Symbols are ordered after numbers
NIL

$ (ORDERP 'DOG 'CAT)                  ;This example assumes "DOG"
T                                       ;is introduced before "CAT"

$ (ORDERP 'CAT 'DOG)
NIL
```

5.6 LOGICAL FUNCTIONS

The logical functions perform Boolean operations on the truth values of their arguments. Every muLISP data object has a truth value. The atom NIL has the truth value false. All other objects have the truth value true.

NOT [*object*] Function

If *object* is the atom NIL, (NOT *object*) returns T, otherwise it returns NIL.

Since the atom NIL represents both the empty list and the truth value false, the functions NULL and NOT always return identical values. However for program readability, it is better to use NULL when testing for the empty list and to use NOT when testing for the truth value false.

```
(DEFUN NOT (OBJ)
  (EQ OBJ NIL) )

$ (NOT NIL)                          ;NIL has truth value false
T

$ (NOT '())                          ;() has truth value false
T

$ (NOT 'FOO)                ;NonNIL symbols have truth value true
NIL

$ (NOT '(A . B))                     ;Conses have truth value true
NIL

$ (NOT (EQ 'DOG 'CAT))               ;DOG is not EQ to CAT
T
```

AND [$form_1$, $form_2$, ..., $form_n$] Special form

(AND $form_1$ $form_2$ ... $form_n$) evaluates each form in turn until one of them evaluates to NIL or all the forms are evaluated. If a form evaluates to NIL, then AND returns NIL; otherwise it returns the value of $form_n$. The call (AND) returns T, an identity for the AND operator. (AND $form_1$ $form_2$ ... $form_n$) is equivalent to the call

(IF $form_1$ (IF $form_2$ (IF ... (IF $form_{n-1}$ $form_n$))))

Note that a form is evaluated if and only if all the preceding forms have evaluated to nonNIL values.

```
(DEFMACRO AND LST
  ((NULL LST))
  ((NULL (CDR LST))
    (CAR LST) )
  (LIST 'IF (CAR LST)
        (MACROEXPAND (CONS 'AND (CDR LST)))) )

$ (AND (EQ 'DOG 'CAT) (< 2 3))     ;One comparison is false
NIL

$ (AND (EQ 'DOG 'DOG) (< 2 3))   ;Both comparisons are true
T

$ (AND (ATOM 'DOG) (MEMBER 'DOG '(CAT DOG COW)))
(DOG COW)                          ;AND returns last value

$ (AND (SYMBOLP 5) (PRINT 'HELLO))     ;Since test is false,
NIL                                    ;(PRINT HELLO) is not evaluated
```

OR [*form*$_1$, *form*$_2$, ..., *form*$_n$] Special form

(OR *form*$_1$ *form*$_2$ ... *form*$_n$) evaluates each form in turn until one of them evaluates to a nonNIL value or all the forms are evaluated. If a form is nonNIL, then OR returns that nonNIL value; otherwise it returns NIL. The call (OR) returns NIL, the identity for the OR operator. (OR *form*$_1$ *form*$_2$ ... *form*$_n$) is equivalent to the call

(COND (*form*$_1$) (*form*$_2$) ... (*form*$_n$))

Note that a form is evaluated if and only if all the preceding forms have evaluated to NIL.

```
(DEFMACRO OR LST
  ((NULL LST) NIL)
  (CONS 'COND
    (MAPCAR '(LAMBDA (FORM) (LIST FORM)) LST)) )

$ (OR (EQ 'DOG 'CAT) (< 2 3))        ;One comparison is true
T

$ (OR (EQ 'DOG 'CAT) (< 3 2))     ;Neither comparison is true
NIL

$ (SETQ FOO 7)                                ;Initialize FOO
7

$ (OR (ZEROP FOO) (/ 35 FOO))          ;OR returns last value
5

$ (SETQ FOO 0)                                ;Initialize FOO
0

$ (OR (ZEROP FOO) (/ 35 FOO))
T                                  ;No "Zero Divide" error break
```

5.7 ASSIGNMENT FUNCTIONS

Assignment functions are normally used to assign values to program variables. For instance, they permit the values of a function's formal arguments to be modified *without* the need for a recursive function call. Thus in some situations they can significantly improve a program's execution speed. Although powerful, indiscriminate use of the assignment functions can result in unstructured LISP programs that are difficult to understand and debug.

If a variable is not a local variable (i.e. not a formal parameter of the currently executing function), assignments to the variable will remain in effect even after the function is exited. As with the modifier functions (see Section 5.3), this phenomenon is called a side-effect of the function.

Although SET is the most primitive assignment function, SETQ is used far more often than SET because assignments are usually made to a variable rather than to the value of a variable. SETQ will be familiar to programmers of more conventional languages where the assignment statement is used pervasively.

SET [*symbol*, *object*] Function

If *symbol* is a nonNIL symbol, (SET *symbol object*) replaces the value cell of *symbol* with a pointer to *object* and returns *object*. If *symbol* is NIL, SET returns object without changing the value of NIL. (Note that the value of the symbol T is *not* protected.) If *symbol* is not a symbol, SET generates a "Nonsymbolic Argument" error break.

RPLACA and SET are identical as far as their effect on their first argument. However, RPLACA returns its first argument, whereas SET returns its second argument. As a general rule, SET should be used if the first argument is a symbol; RPLACA should be used if the first argument is a cons.

```
(DEFUN SET (SYM OBJ)
  ((SYMBOLP SYM)
    ((NULL SYM) OBJ)
    (RPLACA SYM OBJ)
    OBJ )
  (BREAK (LIST 'SET SYM OBJ)
         "Nonsymbolic Argument") )

$ (SET 'FOO '(A B C))                    ;Assign FOO to list
(A B C)

$ FOO                                    ;Current value of FOO
(A B C)

$ (SET 'PET 'DOG)                        ;Assign PET the value DOG
DOG

$ (SET PET 'ROVER)                       ;Assign the value of PET (i.e. DOG)
ROVER                                    ;the value ROVER

$ DOG                                    ;The value of DOG is now ROVER
ROVER

$ PET                                    ;The value of PET is still DOG
DOG
```

SETQ [$symbol_1$, $form_1$, ..., $symbol_n$, $form_n$] Special form

(SETQ *symbol form*) evaluates form, sets the value of *symbol* to result, and returns the result. Note that SETQ is a special form and that *symbol* is *not* evaluated. If *symbol* is not a symbol, SETQ generates a "Nonsymbolic Argument" error break.

If SETQ is given more than two arguments, the form evaluation and assignments are made sequentially. If there are an odd number of arguments, the last symbol is set to NIL. SETQ returns the value assigned to the last symbol.

```
(DEFMACRO SETQ (SYM OBJ)
  (LIST 'SET (LIST 'QUOTE SYM) OBJ) )

$ (SETQ FOO '(A B C))                    ;Assign FOO to list
(A B C)

$ FOO                                    ;Current value of FOO
(A B C)

$ (SETQ FOO (CDR FOO))                   ;Assign FOO to cdr of list
(B C)

$ FOO                                    ;Current value of FOO
(B C)

$ (SETQ NUM 5)                           ;Assign NUM to 5
5

$ (SETQ NUM (+ 3 4) SQR (* NUM NUM))
49                          ;Assigns NUM to 7 and SQR to 7*7

$ NUM                                    ;The value of NUM is now 7
7

$ SQR                                    ;The value of SQR is 49 (not 25)
49
```

PSETQ [$symbol_1$, $form_1$, ..., $symbol_n$, $form_n$] Special form

The effect of PSETQ on its arguments is identical to SETQ except that all the forms are evaluated before making any assignments to symbols. The name "PSETQ" is derived from the phrase "Parallel SETQ".

As the last example shows, SETQ can be used to efficiently interchange the value of two variables without having to introduce an intermediary.

```
$ (SETQ NUM 5)                          ;Assign NUM to 5
5

$ (PSETQ NUM (+ 3 4) SQR (* NUM NUM))
25                          ;Assigns NUM to 7 and SQR to 5*5

$ NUM                                   ;The value of NUM is now 7
7

$ SQR                                   ;The value of SQR is 25 (not 49)
25


$ (PSETQ NUM SQR SQR NUM)               ;Swap value of NUM & SQR
7

$ NUM                                   ;Value of NUM is now 25
25

$ SQR                                   ;Value of SQR is now 7
7
```

SETF [$place_1$, $form_1$, ... $place_n$, $form_n$] Library Macro

If *place* is a valid place form, (SETF *place form*) evaluates *form*, stores the result in *place*, and returns the result. Otherwise, SETF generates an "Invalid Place Form" error break. If SETF is given more than two arguments, the form evaluation and assignments are made sequentially. If there are an odd number of arguments, the last place form is set to NIL. SETF returns the value assigned to the last place form.

Symbols are valid place forms. Lists beginning with one of these selector functions are valid place forms: CAR, CDR, CAAR, CADR, ..., CDDDDR, FIRST, SECOND, THIRD, ..., TENTH, REST, NTH, GET, MACRO-FUNCTION, SYMBOL-VALUE, SYMBOL-PLIST, and SYMBOL-FUNCTION. Macro calls that expand into valid place forms are valid place forms.

```
(DEFMACRO SETF LST
  ((NULL LST) NIL)
  ((NULL (CDDR LST))
    ((ATOM (CAR LST))
      (CONS 'SETQ LST) )
    (MAKE-SET-FORM (CAR LST) (CADR LST)) )
  (CONS 'PROGN (SETF-AUX LST)) )

(DEFUN SETF-AUX (LST)
  ((NULL LST) NIL)
  ((ATOM (CAR LST))
    (CONS (LIST 'SETQ (CAR LST) (CADR LST))
          (SETF-AUX (CDDR LST))) )
  (CONS (MAKE-SET-FORM (CAR LST) (CADR LST))
        (SETF-AUX (CDDR LST))) )

$ (SETQ FOO '(A B C))                    ;Initialize FOO
(A B C)

$ (SETF (SECOND FOO) 5)          ;2nd element of FOO set to 5
5

$ FOO                                    ;Newly modified FOO
(A 5 C)
```

POP [*symbol*] Special form

(POP *symbol*) returns the "top" (i.e car) of the stack (i.e. list) named *symbol* and updates the value of *symbol* to the rest (i.e. cdr) of the stack. If *symbol* is not a symbol, a "Nonsymbolic Argument" error break occurs. If the value of *symbol* is not a cons, POP returns NIL.

This special form provides the LISP analog of the machine language pop instruction used to fetch information from the top of a stack.

```
(DEFMACRO POP (SYM)
  (LIST 'PROG1
        (LIST 'CAR SYM)
        (LIST 'SETQ SYM (LIST 'CDR SYM))) )

$ (SETQ STACK-LIST '(A B C D E F))        ;Initialize stack
(A B C D E F)

$ (POP STACK-LIST)                        ;Pop A off stack
A

$ (POP STACK-LIST)                        ;Pop B off stack
B

$ (POP STACK-LIST)                        ;Pop C off stack
C

$ STACK-LIST                              ;Shortened stack
(D E F)
```

PUSH [*form, symbol*] — Special form
PUSHNEW [*form, symbol, test*] — Library Macro

(PUSH *form symbol*) evaluates *form* and pushes (i.e. conses) the result onto the "stack" named *symbol* and updates the value of *symbol* to the larger stack. PUSH returns the enlarged stack (i.e. list). If *symbol* is not a symbol, a "Nonsymbolic Argument" error break occurs. This special form provides the LISP analog of the machine language push instruction used to save information on top of a stack.

If the value of *form* is *not* a member of the "stack" named *symbol*, (PUSHNEW *form symbol test*) pushes (i.e. conses) the value of *form* onto the stack, updates the value of *symbol* to the larger stack, and returns the enlarged stack. Otherwise, PUSHNEW simply returns the stack. If *symbol* is not a symbol, a "Nonsymbolic Argument" error break occurs.

```
(DEFMACRO PUSH (OBJ SYM)
  (LIST 'SETQ SYM (LIST 'CONS OBJ SYM)) )

(DEFMACRO PUSHNEW (OBJ SYM TEST)
  (LIST 'SETQ SYM (LIST 'ADJOIN OBJ SYM TEST)) )

$ (SETQ STACK-LIST NIL)                    ;Empty stack
NIL

$ (PUSH 'A STACK-LIST)                     ;Push A on stack
(A)

$ (PUSH 'B STACK-LIST)                     ;Push B on stack
(B A)

$ (PUSHNEW 'C STACK-LIST)                  ;Push C on stack
(C B A)

$ (PUSHNEW 'B STACK-LIST)                  ;B already on stack
(C B A)

$ STACK-LIST                               ;Current stack
(C B A)
```

5.8 PROPERTY FUNCTIONS

The property functions are used to manage properties associated with symbols. The cdr cell of a symbol points to a property list containing properties and flags (see Section 5.9). A property list is an ASSOCiation list of property keys paired with property values (see the description of ASSOC in Section 5.1). Since flags are atomic elements of the property list, they can be distinguished from properties.

The property and flag functions facilitate the construction of dynamically extendible data bases from which information can be easily and rapidly extracted.

PUT [*symbol*, *key*, *object*] Function

(PUT *symbol key object*) places on the property list of *symbol* under the indicator *key* the value *object*.

Since PUT (i.e. ASSOC) uses an EQUAL test when searching the property list for a match to *key*, property keys need not be atomic. Note that any previous value under the indicator *key* is automatically discarded.

Since PUT uses a RPLACD operation to add a property value to a property list, care should used when sharing property lists.

```
(DEFUN PUT (SYM KEY OBJ)
  ((OR (NULL SYM) (NOT (SYMBOLP SYM))) NIL)
  ((NULL (ASSOC KEY (CDR SYM)))
    (RPLACD SYM (ACONS KEY OBJ (CDR SYM)))
    OBJ )
  (RPLACD (ASSOC KEY (CDR SYM)) OBJ)
  OBJ )

$ (PUT 'USA 'POPULATION 226000000)
226000000                          ;Population of countries

$ (PUT 'FRANCE 'POPULATION 55000000)
55000000

$ (PUT 'JAPAN 'POPULATION 119000000)
119000000

$ (PUT 'USA 'AREA 3536855)
3536855                        ;Area of countries in square miles

$ (PUT 'FRANCE 'AREA 211208)
211208

$ (PUT 'JAPAN 'AREA 142726)
142726
```

GET [*symbol, key*] **Function**

(GET *symbol key*) returns the property value associated with *symbol* under the indicator *key*. If no such value exists, GET returns NIL.

The examples for GET assume that the examples on the previous page for PUT have been performed.

```
(DEFUN GET (SYM KEY)
  ((NULL (ASSOC KEY (CDR SYM))) NIL)
  (CDR (ASSOC KEY (CDR SYM))) )

$ (GET 'USA 'POPULATION)
226000000                              ;USA's population

$ (GET 'AUSTRALIA 'POPULATION)
NIL                          ;Australia's population is unknown


                                       ;Population density function
$ (DEFUN POPULATION-DENSITY (COUNTRY)
    (/ (GET COUNTRY 'POPULATION)
       (GET COUNTRY 'AREA)) )
POPULATION-DENSITY

$ (POPULATION-DENSITY 'FRANCE)
260.4067796                          ;French per square mile

$ (POPULATION-DENSITY 'JAPAN)
833.7647058                          ;Japanese per square mile
```

REMPROP [*symbol, key*] Function

(REMPROP *symbol key*) removes from *symbol*'s property list the property associated with *key*. REMPROP returns the old property value if one was found, otherwise it returns NIL.

Since REMPROP uses a RPLACD operation to remove an element from a property list, care should used when sharing property lists.

The examples for REMPROP assume that the examples given earlier for PUT have been performed.

```
(DEFUN REMPROP (SYM KEY)
  ((ATOM (CDR SYM)) NIL)
  ((EQUAL (CAADR SYM) KEY)
    (SETQ KEY (CDADR SYM))
    (RPLACD SYM (CDDR SYM))
    KEY )
  (REMPROP (CDR SYM) KEY) )

$ (REMPROP 'USA 'AREA)                    ;Remove property
3536855

$ (GET 'FRANCE 'AREA)                     ;France's area
211208

$ (GET 'USA 'AREA)                        ;USA's area is unknown
NIL
```

5.9 FLAG FUNCTIONS

The flag functions, like the property functions, can be used to store information on the property list of a symbol. The symbol is merely flagged as having an attribute by placing the attribute (usually a symbol) on the property list.

FLAG [*symbol*, *attribute*] Function
FLAGP [*symbol*, *attribute*] Function

(FLAG *symbol attribute*) flags *symbol* with *attribute* by making *attribute* the first element of the *symbol*'s property list. *attribute* is only added to *symbol*'s property list if not already present.

If *attribute* is a member of *symbol*'s property list, (FLAGP *symbol attribute*) returns a nonNIL value (i.e. the property list beginning with *attribute*, otherwise it returns NIL.

```
(DEFUN FLAG (SYM ATTRIB)
  ((OR (NULL SYM) (NOT (SYMBOLP SYM))) NIL)
  ((FLAGP SYM ATTRIB) ATTRIB)
  (RPLACD SYM (CONS ATTRIB (CDR SYM)))
  ATTRIB )

(DEFUN FLAGP (SYM ATTRIB)
  (MEMBER ATTRIB (CDR SYM) 'EQUAL) )

$ (FLAG 'JOHN 'MALE)                    ;Flag John as a male
MALE

$ (FLAG 'SUE 'FEMALE)                   ;Flag Sue as a female
FEMALE

$ (FLAGP 'JOHN 'MALE)                       ;John is a male
(MALE)

$ (FLAGP 'SUE 'MALE)                        ;Sue is not a male
NIL
```

REMFLAG [*symbol, attribute*] Function

(REMFLAG *symbol attribute*) removes *attribute* from the property list of *symbol* and returns T. If the attribute is not found, REMFLAG returns NIL.

```
(DEFUN REMFLAG (SYM ATTRIB)
  ((ATOM (CDR SYM)) NIL)
  ((EQUAL ATTRIB (CADR SYM))
    (RPLACD SYM (CDDR SYM))
    T )
  (REMFLAG (CDR SYM) ATTRIB) )
```

```
$ (FLAG 'JAN 'MALE)                    ;Flag Jan as a male
MALE

$ (FLAG 'JAN 'TALL)                    ;Flag Jan as tall
TALL

$ (REMFLAG 'JAN 'MALE)                 ;Remove male flag
T

$ (FLAGP 'JAN 'MALE)                   ;Jan's sex unknown
NIL

$ (FLAGP 'JAN 'TALL)                   ;Jan is still tall
(TALL)
```

5.10 DEFINITION PRIMITIVES

These primitives are used to manipulate function definitions. When a function is defined, the definition is automatically pseudo-compiled into an extremely dense form called D-code or distilled code. Pseudo-compiling the definition results in a two-fold increase in code density and a 20% improvement in interpretation speed.

The inverse process of de-compiling a function definition into a linked list also occurs automatically when GETD is used to retrieve a definition. Thus both pseudo-compilation and de-compilation are invisible to the user, and the interactive, incremental nature of LISP is not compromised despite the great improvement in efficiency and code density.

GETD [*symbol*, *flag*]	Function
SPECIAL	Keyword
LAMBDA	Keyword
NLAMBDA	Keyword
MACRO	Keyword

If *symbol* is a user-defined function, (GETD *symbol*) decompiles and returns the definition of *symbol* in linked list form. If *symbol* is a primitive function or special form, (GETD *symbol*) returns the physical memory address (in the 8086 code segment) of the machine language routine. GETD returns the offset of the routine in the code segment. If *symbol* is not a function or special form, (GETD *symbol*) returns NIL.

If *flag* is nonNIL, (GETD *symbol flag*) returns SPECIAL if *symbol* is a special form, LAMBDA if *symbol* is an eval function, NLAMBDA if *symbol* is a no-eval function, MACRO if *symbol* is a macro function, or NIL if *symbol* is undefined. See Section 5.16 for an explanation of the different types of functions.

If a program only needs to know if *symbol* is a defined function, it is faster to call GETD with *flag* nonNIL since no consing is required.

```
(DEFUN GETD (SYM FLAG)
  ((NOT (SYMBOLP SYM)) NIL)
  ((undefined SYM) NIL).
  ((NULL FLAG)
    ((machine language routine SYM)
      (the memory address of the routine) )
    (the S-expression equivalent of the D-code
       defining the function SYM) )
  ((special form SYM) 'SPECIAL)
  ((eval function SYM) 'LAMBDA)
  ((no-eval function SYM) 'NLAMBDA)
  ((macro function SYM) 'MACRO) )

$ (GETD 'CONS)                    ;Address of CONS routine
2863

$ (GETD 'CONS T)                  ;CONS is an eval function
LAMBDA
```

PUTD [*symbol*, *definition*] Function

If *definition* is a lambda-list, (PUTD *symbol definition*) pseudo-compiles *definition* into D-code, modifies *symbol*'s function definition cell to point to the compiled D-code, and returns *symbol*. Quoted nonatomic objects (i.e. constants) within a function definition are not pseudo-compiled but remain stored as a linked list. Therefore, the interpreter does not have to de-compile such constants each time they are encountered, and constants can contain conses with nonNIL atomic cdr cells.

If *definition* is zero or a positive integer less than 65536, (DEFUN *symbol definition*) modifies *symbol*'s function cell to *definition*, and returns *symbol*. This makes it possible to link to user-defined, machine language routines. See Chapter 9 for details on linking to machine language routines.

The variable *CONDENSE* controls the condensing of D-code while pseudo-compiling. The variable MACROEXPAND controls the expansion of macros while pseudo-compiling (see Section 5.16).

```
(DEFUN PUTD (SYM DEFN)
  ((OR (NOT (SYMBOLP SYM)) (SYMBOLP DEFN)) NIL)
  ((NUMBERP DEFN)
    ((AND (INTEGERP DEFN) (PLUSP DEFN))
      (store DEFN in SYM's function cell)
      SYM ) )
  (pseudo-compile DEFN into D-code)
  (store a pointer to the D-code in SYM's
     function cell)
  SYM )

$ (PUTD 'SQUARE '(LAMBDA (NUM) (* NUM NUM)))
SQUARE                          ;Define function using PUTD

$ (SQUARE 5)                                ;Call function
25
```

MOVD [*symbol*$_1$, *symbol*$_2$] Function

(MOVD *symbol*$_1$ *symbol*$_2$) copies *symbol*$_1$'s function definition cell to *symbol*$_2$'s and returns *symbol*$_2$. In cases where MOVD is sufficient, it should be used instead of a combination of GETD and PUTD since no more pseudo-compilation is required.

The examples show how MOVD can be used to make more mnemonic symbols for some of the primitive LISP selector functions without sacrificing execution efficiency.

```
(DEFUN MOVD (SYM1 SYM2)
  ((AND (SYMBOLP SYM1) (SYMBOLP SYM2))
    (replace the contents of SYM2's function
        cell with the contents of SYM1's
        function cell)
     SYM2 ) )

$ (MOVD 'CAR 'FIRST)                    ;Define FIRST
FIRST

$ (MOVD 'CDR 'REST)                     ;Define REST
REST

$ (MOVD 'CADR 'SECOND)                  ;Define SECOND
SECOND

$ (MOVD 'CADDR 'THIRD)                  ;Define THIRD
THIRD

$ (MOVD 'CADDDR 'FOURTH)                ;Define FOURTH
FOURTH

$ (FIRST '(A B C D))                    ;Call FIRST
A
```

REMD [*symbol*] — Function
FMAKUNBOUND [*symbol*] — Library Function

(REMD *symbol*) modifies *symbol*'s function definition cell to make *symbol* undefined. REMD returns *symbol*.

The definition of FMAKUNBOUND is *identical* to that of REMD and is included in muLISP solely for Common LISP compatibility.

Since symbols can only be collected by the garbage collector if they are undefined, REMD is often used for for this purpose. See Chapter 4 for information on how to garbage collect symbols.

```
(DEFUN REMD (SYM)
  ((SYMBOLP SYM)
    (change SYM's function cell to be
        undefined) ) )

$ (DEFUN RCONS (SYM1 SYM2) (CONS SYM2 SYM1))
RCONS                              ;Define RCONS function

$ (RCONS 'DOG 'CAT)                          ;Call RCONS
(CAT . DOG)

$ (REMD 'RCONS)                  ;Remove RCONS's definition
RCONS

$ (GETD 'RCONS)                    ;RCONS now undefined
NIL

$ (RCONS 'DOG 'CAT)                          ;Call RCONS
Undefined Function Break: (RCONS 'DOG 'CAT)
```

DEFUN [*symbol*, *arglist*, $form_1$, ..., $form_n$] Special form
DEFUN [*symbol*, (*ftype arglist* $form_1$ ... $form_n$)] Special form
DEFUN [*symbol*, *address*] Special form

The function defining special form DEFUN can be called in either of three ways: The first way is for defining eval functions. The second way is for defining no-eval functions and macros. The third way is for linking to user-defined, machine-language routines.

(DEFUN *symbol arglist* $form_1$ ... $form_n$) pseudo-compiles *arglist* and the forms into D-code assuming an eval or LAMBDA function type. DEFUN then modifies *symbol*'s function definition cell to point to the resulting D-code, and returns *symbol*.

If *ftype* is the symbol LAMBDA, NLAMBDA, or MACRO, the call (DEFUN *symbol* (*ftype arglist* $form_1$ ... $form_n$)) pseudo-compiles *arglist* and the forms into D-code assuming an eval, no-eval, or macro function depending on *ftype*. DEFUN then modifies *symbol*'s function cell to point to the resulting D-code, and returns *symbol*.

If *address* is zero or a positive integer less than 65536, (DEFUN *symbol address*) modifies *symbol*'s function cell to *address*, and returns *symbol*. See Chapter 9 for details on linking to machine language routines.

Although it is defined in terms of the more primitive function PUTD, DEFUN is usually a more convenient means of defining functions. Examples of DEFUN appear throughout this chapter.

```
(DEFMACRO DEFUN (SYM . BODY)
  ((ATOM BODY) NIL)
  ((NUMBERP (CAR BODY))
    (LIST 'PUTD (LIST 'QUOTE SYM) (CAR BODY)) )
  ((MEMBER (CAAR BODY) '(LAMBDA NLAMBDA MACRO))
    (LIST 'PUTD (LIST 'QUOTE SYM)
                (CONS 'QUOTE BODY)) )
  (LIST 'PUTD (LIST 'QUOTE SYM)
              (LIST 'QUOTE (CONS 'LAMBDA BODY))) )
```

DEFMACRO [*symbol*, *arglist*, *form*$_1$, ..., *form*$_n$] Special form

(DEFMACRO *symbol arglist form*$_1$ *form*$_2$ ... *form*$_n$) destructures and pseudo-compiles *arglist* and all the forms into D-code, modifies *symbol*'s function definition cell to point to the resulting D-code, and returns *symbol*.

Unlike DEFUN, DEFMACRO provides for the destructuring of *arglist*. This facility means that arguments are assigned to the symbols in *arglist* based on the linked list structure of *arglist*. If *arglist* is a symbol, the actual arguments in the macro call are assigned to the symbol. Otherwise the car of the arguments are recursively assigned to the car of *arglist* and the cdr of the arguments are assigned to the cdr of *arglist*.

The formal definition of DEFUN on the previous page is an example of the use of DEFMACRO's destructuring capability.

```
(PUTD DEFMACRO '(MACRO BODY
  (POP BODY)
  (LIST 'PUTD
        (LIST 'QUOTE (CAR BODY))
        (LIST 'QUOTE (LIST 'MACRO 'BODY
           (CONS (LIST* 'LAMBDA
                        (Leaves (CADR BODY)) (CDDR BODY))
                 (Destructure (CADR BODY) '(CDR BODY]

(DEFUN Leaves (TREE)
  ((ATOM TREE) (LIST TREE) )
  ((NULL (CDR TREE))
    (Leaves (CAR TREE)) )
  (NCONC (Leaves (CAR TREE)) (Leaves (CDR TREE))))

(DEFUN Destructure (TREE FORM)
  ((ATOM TREE) (LIST FORM) )
  ((NULL (CDR TREE))
    (Destructure (CAR TREE) (LIST 'CAR FORM)) )
  (NCONC (Destructure (CAR TREE) (LIST 'CAR FORM))
    . (Destructure (CDR TREE) (LIST 'CDR FORM))) )
```

CONDENSE: NIL Control variable

As a function definition is being pseudo-compiled, a search is made to find existing D-code equivalent to each subexpression in the definition. If such equivalent D-code is found, it is used instead of creating more D-code, thereby saving memory. This process is called condensing D-code.

If *CONDENSE* is NIL, the condensing search is limited to the definition's own D-code. Although it increases the D-code space memory usage, limiting the condensing search space greatly reduces the time required to load large source files.

If *CONDENSE* is nonNIL, the condensing search spans the entire existing D-code. This full condensing reduces the amount of memory required to store a given set of function definitions at a cost of increased time required to load large source files. Using a small set of generic names (e.g. NUM1, NUM2, LST, OBJ, ATM, etc.) in the formal argument lists of user defined functions can significantly increase the code density benefits derived from full condensing.

When a user-defined function is redefined using PUTD or MOVD, or has its definition removed using REMD, the D-code used to store the definition is automatically reclaimed if full condensing has *not* been performed (i.e. if *CONDENSE* has been NIL) since the function was defined. Thus, it is wise to have *CONDENSE* NIL during interactive program development when functions are frequently redefined.

5.11 PRINT NAME FUNCTIONS

The character and string manipulation functions are useful for text and language processing applications. They provide a rich set of operations for comparing, searching, extracting, and transforming the print names of both symbols and numbers.

5.11.1 Character Functions

UPPER-CASE-P [*symbol*] Function
LOWER-CASE-P [*symbol*] Function

If the first character of *symbol*'s print name is an uppercase character, (UPPER-CASE-P *symbol*) returns T, otherwise it returns NIL. The uppercase characters are the letters *A* through *Z*.

If the first character of *symbol*'s print name is a lowercase character, (LOWER-CASE-P *symbol*) returns T, otherwise it returns NIL. The lowercase characters are the letters *a* through *z*.

```
(DEFUN UPPER-CASE-P (SYM)
  ((SYMBOLP SYM)
    (<= 65 (ASCII SYM) 90) ) )

(DEFUN LOWER-CASE-P (SYM)
  ((SYMBOLP SYM)
    (<= 97 (ASCII SYM) 122) ) )

$ (UPPER-CASE-P "D")
T

$ (UPPER-CASE-P "d")
NIL

$ (LOWER-CASE-P "D")
NIL

$ (LOWER-CASE-P "d")
T
```

ALPHA-CHAR-P [*symbol*] Function
NUMERIC-CHAR-P [*symbol*] Function
ALPHANUMERICP [*symbol*] Function

If the first character of *symbol*'s print name is an alphabetic character, (ALPHA-CHAR-P *symbol*) returns T, otherwise it returns NIL. The alphabetic characters are the letters *A* through *Z* and *a* through *z*.

If the first character of *symbol*'s print name is a numeric character, (NUMERIC-CHAR-P *symbol*) returns T, otherwise it returns NIL. The numeric characters are the digits *0* through *9*.

If the first character of *symbol*'s print name is an alphabetic or numeric character, (ALPHANUMERICP *symbol*) returns T, otherwise it returns NIL.

```
(DEFUN ALPHA-CHAR-P (SYM)
  (OR (UPPER-CASE-P SYM) (LOWER-CASE-P SYM)) )

(DEFUN NUMERIC-CHAR-P (SYM)
  ((SYMBOLP SYM)
    (<= 48 (ASCII SYM) 57) ) )

(DEFUN ALPHANUMERICP (SYM)
  (OR (ALPHA-CHAR-P SYM) (NUMERIC-CHAR-P SYM)) )

$ (ALPHA-CHAR-P "D")
T

$ (ALPHA-CHAR-P "3")
NIL

$ (NUMERIC-CHAR-P "D")
NIL

$ (NUMERIC-CHAR-P "3")
T

$ (ALPHANUMERICP "D")
T
```

DIGIT-CHAR-P [*symbol*, *radix*] Function

If *radix* is an integer between 2 and 36 inclusive and the first character of *symbol*'s print name is a character whose numeric weight is less than *radix*, (DIGIT-CHAR-P *symbol radix*) returns the numeric weight of the character. *radix* defaults to 10. If the character is not less than *radix*, DIGIT-CHAR-P returns NIL.

The numeric weights of the digits *0* through *9* are the integers 0 through 9. The weights of the letters *A* through *Z* (and also the letters *a* through *z*) are the integers 10 through 35.

```
(DEFUN DIGIT-CHAR-P (SYM RADIX)
  ((AND (INTEGERP RADIX) (<= 2 RADIX 36))
    ((NUMERIC-CHAR-P SYM)
      (SETQ SYM (- (ASCII SYM) 48))
      ((< SYM RADIX) SYM) )
    ((ALPHA-CHAR-P SYM)
      (SETQ SYM (- (ASCII (CHAR-UPCASE SYM)) 55))
      ((< SYM RADIX) SYM) ) )
  (DIGIT-CHAR-P SYM 10) )

$ (DIGIT-CHAR-P "5")                    ;5 is the weight of "5"
5

$ (DIGIT-CHAR-P "5" 2)                  ;"5" is not a binary digit
NIL

$ (DIGIT-CHAR-P "D")                    ;"D" is not a decimal digit
NIL

$ (DIGIT-CHAR-P "D" 16)                 ;13 is the weight of "D"
13
```

CHAR= [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR< [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR> [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR<= [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR>= [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function
CHAR/= [$symbol_1$, $symbol_2$, ..., $symbol_n$] Function

These functions perform a lexicographical comparison on the first character of the print name of their arguments taking into account the distinction between upper and lower case letters. CHAR= performs an equal test, CHAR< performs a less than test, CHAR> performs a greater than test, CHAR<= performs a less than or equal test, CHAR>= performs a greater than or equal test, and CHAR/= performs a not equal test. If the comparison is true, these functions return T, otherwise they return NIL.

```
(DEFUN CHAR< (SYM1 SYM2)
  ((AND (SYMBOLP SYM1) (SYMBOLP SYM2))
    (< (ASCII SYM1) (ASCII SYM2)) ) )

(DEFUN CHAR<= (SYM1 SYM2)
  ((AND (SYMBOLP SYM1) (SYMBOLP SYM2))
    (<= (ASCII SYM1) (ASCII SYM2)) ) )

$ (CHAR< "D" "C")                    ;"D" ordered after "C"
NIL

$ (CHAR< "C" "D" "G")                ;Characters in order
T

$ (CHAR< "D" "c")          ;Uppercase ordered before lowercase
T
```

CHAR-EQUAL [$symbol_1$, $symbol_2$,..., $symbol_n$] Function
CHAR-LESSP [$symbol_1$, $symbol_2$,..., $symbol_n$] Function
CHAR-GREATERP [$symbol_1$, $symbol_2$,..., $symbol_n$] Function
CHAR-NOT-GREATERP [$symbol_1$, $symbol_2$,..., $symbol_n$] Function
CHAR-NOT-LESSP [$symbol_1$, $symbol_2$,..., $symbol_n$] Function
CHAR-NOT-EQUAL [$symbol_1$, $symbol_2$,..., $symbol_n$] Function

These functions perform a lexicographical comparison on the first character of the print name of their atomic arguments making no distinction between upper and lower case letters. CHAR-EQUAL performs an equal test, CHAR-LESSP performs a less than test, CHAR-GREATERP performs a greater than test, CHAR-NOT-GREATERP performs a less than or equal test, CHAR-NOT-LESSP performs a greater than or equal test, and CHAR-NOT-EQUAL performs a not equal test. If the comparison is true, these functions return T, otherwise they return NIL.

```
(DEFUN CHAR-LESSP (SYM1 SYM2)
  ((AND (SYMBOLP SYM1) (SYMBOLP SYM2))
    (< (ASCII (CHAR-UPCASE SYM1))
       (ASCII (CHAR-UPCASE SYM2))) ) )

(DEFUN CHAR-NOT-GREATERP (SYM1 SYM2)
  ((AND (SYMBOLP SYM1) (SYMBOLP SYM2))
    (<= (ASCII (CHAR-UPCASE SYM1))
        (ASCII (CHAR-UPCASE SYM2))) ) )

$ (CHAR-LESSP "D" "C")                    ;"D" ordered after "C"
NIL

$ (CHAR-LESSP "C" "D" "G")                ;Characters in order
T

$ (CHAR-LESSP "D" "c")                    ;Case ignored
NIL
```

ASCII [*atom*] Function

(ASCII *atom*) returns the ASCII equivalent of *atom*. If *atom* is a symbol, ASCII returns the integer value of the ASCII code used to denote the first character in *atom*'s print name. If *atom* is a nonnegative integer less than 256, ASCII returns the symbol whose print name is the ASCII character *atom*. Otherwise ASCII returns NIL.

ASCII is especially useful for generating the ASCII control characters necessary to execute screen commands for various terminals and computers from within muLISP.

```
(DEFUN ASCII (ATM)
  ((SYMBOLP ATM)
    ((ZEROP (LENGTH ATM)) NIL)
    (the integer equivalent of the first character
        in the print name of ATM) )
  ((AND (INTEGERP ATM) (< -1 ATM 256))
    (the symbol whose print name is the single
        character equivalent to ATM) ) )

$ (ASCII "A")                          ;ASCII code for "A"
65

$ (ASCII " ")                          ;ASCII code for blank
32

$ (ASCII 65)                  ;Character whose ASCII code is 65
A

$ (ASCII "4")                          ;ASCII code for "4"
52
```

CHAR-CODE [*symbol*] Library Function
CODE-CHAR [*n*] Library Function

(CHAR-CODE *symbol*) returns the integer value of the ASCII code used to denote the first character in *symbol*'s print name. Otherwise CHAR-CODE returns NIL.

If *n* is a nonnegative integer less than 256, (CODE-CHAR *n*) returns the symbol whose print name is the ASCII character *n*. Otherwise CODE-CHAR returns NIL.

CHAR-CODE and CODE-CHAR are easily defined in terms of the more general function ASCII. They are included in muLISP solely for Common LISP compatibility.

```
(DEFUN CHAR-CODE (SYM)
  ((SYMBOLP SYM)
    (ASCII SYM) ) )

(DEFUN CODE-CHAR (N)
  ((AND (INTEGERP N) (<= 0 N 255))
    (ASCII N) ) )

$ (CHAR-CODE "A")                    ;ASCII code for "A"
65

$ (CHAR-CODE " ")                    ;ASCII code for blank
32

$ (CHAR-CODE 65)                     ;65 is not a symbol
NIL

$ (CODE-CHAR 65)          ;Character whose ASCII code is 65
A

$ (CODE-CHAR "4")                    ;"4" is not an integer
NIL
```

CHAR-UPCASE [*symbol*] Function
CHAR-DOWNCASE [*symbol*] Function

(CHAR-UPCASE *symbol*) returns the symbol whose print name is the first character of *symbol*'s print name in uppercase. (CHAR-DOWNCASE *symbol*) returns the symbol whose print name is the first character of *symbol*'s print name in lowercase. If *symbol* is not a symbol, CHAR-UPCASE and CHAR-DOWNCASE return NIL.

```
(DEFUN CHAR-UPCASE (SYM)
  ((SYMBOLP SYM)
    ((LOWER-CASE-P SYM)
      (ASCII (- (ASCII SYM) 32)) )
    (CHAR SYM 0) ) )

(DEFUN CHAR-DOWNCASE (SYM)
  ((SYMBOLP SYM)
    ((UPPER-CASE-P SYM)
      (ASCII (+ (ASCII SYM) 32)) )
    (CHAR SYM 0) ) )

$ (CHAR-UPCASE "ABC")
A

$ (CHAR-UPCASE "abc")
A

$ (CHAR-DOWNCASE "ABC")
\a

$ (CHAR-DOWNCASE "abc")
\a
```

))

5.11.2 String Functions

The print name of symbols are stored explicitly as character strings. The print name of numbers must be derived implicitly from their value which is stored in binary. When called with numerical arguments, the string functions automatically create a character string equivalent to the numbers' value based on the current output radix base (see *PRINT-BASE* in Section 5.14). Thus, it is important to note that the print name of a number, as seen by the string functions, changes when the output radix base changes.

UNPACK [*atom*] Function

(UNPACK *atom*) returns a list of symbols whose one character print names are the characters in the printed representation of *atom*. If *atom* is nonatomic, UNPACK returns NIL. UNPACK is called EXPLODE in some dialects of LISP.

```
(DEFUN UNPACK (ATM)
  ((SYMBOLP ATM)
    (a list of symbols whose print names consist
        of the characters in ATM's print name) )
  ((NUMBERP ATM)
    (a list of symbols whose print names consist
        of the digits in the printed represented
        of ATM) ) )

$ (UNPACK 'ABCDE)                          ;List of char symbols
(A B C D E)

$ (SETQ FOO -216)                          ;Set FOO to -216
-216

$ (SETQ *PRINT-BASE* 16)                   ;Set output radix to 16
10

$ (UNPACK FOO)                             ;List of hexadecimal numerals
(- \0 D \8)
```

PACK [*list*] Function
PACK* [$atom_1$, $atom_2$, ..., $atom_n$] Function

(PACK *list*) returns the symbol whose print name consists of the concatenated print names of the atoms in *list*. (PACK* $atom_1$ $atom_2$ ... $atom_n$) returns the symbol whose print name consists of the concatenated print names of $atom_1$ through $atom_n$. PACK* is merely a no-spread version of PACK (i.e. it takes an arbitrary number of atoms rather than a list of atoms).

The current output radix base is used to determine the print name of numbers. Note that PACK always returns a symbol, even if its print name contains only digits. PACK is called COMPRESS or IMPLODE in some dialects of LISP.

```
(DEFUN PACK (LST)
  ((ATOM LST) "")
  ((SYMBOLP (CAR LST))
    (the symbol whose print name consists of the
        print name of (CAR LST) concatenated with
        (PACK (CDR LST))) )
  ((NUMBERP (CAR LST))
    (the symbol whose print name consists of the
        digits in the printed representation of
        (CAR LST) concatenated with
        (PACK (CDR LST))) )
  (PACK (CDR LST)) )

(DEFUN PACK* LST
  (PACK LST) )

$ (PACK (LIST 'WARE 'HOUSE))          ;Pack list of symbols
WAREHOUSE

$ (PACK* 'WARE 'HOUSE)                ;Pack symbols
WAREHOUSE
```

SUBSTRING [*atom*, *n*, *m*] Function

If *atom* is either a symbol or a number, *n* and *m* are nonnegative integers, and *n* <= *m*, (SUBSTRING *atom n m*) returns the symbol whose print name consists of the *n*th through *m*th characters of *atom*'s print name, where the first character of the print name is the *zeroth* character.

All values of *n* less than 0 are equivalent to *n* being equal to 0. If *m* is omitted, is not a nonnegative integer, or is greater than the number of characters in *atom*'s print name, it is equivalent to *m* being equal to the number of characters in the print name. If *n* is greater than or equal to the number of characters in the print name or if *n* > *m*, SUBSTRING returns the symbol whose print name is the null string.

Note that SUBSTRING always returns a symbol even though *atom* may be a number.

```
(DEFUN SUBSTRING (ATM N M)
  ((AND (ATOM ATM) (INTEGERP N))
    ((MINUSP N)
      (SUBSTRING ATM 0 M) )
    (PACK (SUBLIST (UNPACK ATM) N M)) ) )

$ (SUBSTRING 'ABCDEFG 2 4)              ;2nd thru 4th letters
CDE

$ (SUBSTRING 'ABCDEFG 2)                ;2nd thru last letters
CDEFG

$ (SUBSTRING 'ABCDEFG 0 4)              ;0th thru 4th letters
ABCDE

$ (SUBSTRING 1000 0)                    ;Convert number to symbol
|1000|
```

CHAR [*atom*, *n*] Function

If *atom* is either a symbol or a number and *n* is a nonnegative integer, (CHAR *atom n*) returns the symbol whose print name is the *n*th character of *atom*'s print name, where the first character of the print name is the *zeroth* character. If *n* is neither zero nor a positive integer or if *atom*'s print name has *n* or fewer characters, CHAR returns NIL.

Note that CHAR always returns a symbol even though *atom* may be a number.

```
(DEFUN CHAR (ATM N)
  (SUBSTRING ATM N N) )

$ (CHAR 'ABCDEFG 3)                    ;Third element
D

$ (CHAR 5432 0)                        ;Zeroth element
\5                          ;CHAR always returns a symbol

$ (CHAR 'ABCDEFG 10)                   ;Beyond end
NIL
```

STRING= [*atom*$_1$, *atom*$_2$, *flag*] — Function
STRING-EQUAL [*atom*$_1$, *atom*$_2$] — Library Macro

If *atom*$_1$'s print name is lexicographically equal to *atom*$_2$'s print name, (STRING= *atom*$_1$ *atom*$_2$) return T, otherwise it returns NIL. If *flag* is nonNIL, (STRING= *atom*$_1$ *atom*$_2$ *flag*) makes the same comparison but with no distinction between upper and lower case letters.

STRING-EQUAL makes the same comparison as STRING= but with no distinction between upper and lower case letters. Thus, STRING-EQUAL provides no additional functionally and is included in muLISP solely for Common LISP compatibility.

```
(DEFUN STRING= (ATM1 ATM2 FLAG)
  ((AND (ATOM ATM1) (ATOM ATM2))
    ((NULL FLAG)
      ((if ATM1's print name is lexicographically
          equal to ATM2's print name) T) )
    ((if ATM1's print name is lexicographically,
       by ignoring case, equal to ATM2's print
       name) T) ) )

(DEFMACRO STRING-EQUAL (ATM1 ATM2)
  (LIST 'STRING= ATM1 ATM2 T) )

$ (STRING= "FAST" "FASTER")          ;Unequal string lengths
NIL

$ (STRING= 100 |100|)                ;Have same print names
T

$ (STRING= "Fast" "FAST")                    ;Case sensitive
NIL

$ (STRING= "Fast" "FAST" T)                  ;Case insensitive
T
```

STRING< [$atom_1$, $atom_2$, *flag*] Function
STRING> [$atom_1$, $atom_2$, *flag*] Function
STRING<= [$atom_1$, $atom_2$, *flag*] Function
STRING>= [$atom_1$, $atom_2$, *flag*] Function
STRING/= [$atom_1$, $atom_2$, *flag*] Function

If $atom_1$'s print name is lexicographically less than $atom_2$'s print name, (STRING< $atom_1$ $atom_2$) returns the position of the first character at which the print names fail to match, otherwise it returns NIL. If *flag* is nonNIL, (STRING< $atom_1$ $atom_2$ *flag*) makes the same comparison with no distinction between upper and lower case letters.

The other four functions also perform a lexicographical comparison and return NIL or the position of the first mismatch. STRING> performs a greater than test, STRING<= performs a less than or equal test, STRING>= performs a greater than or equal test, and STRING/= performs a not equal test.

```
(DEFUN STRING< (ATM1 ATM2 FLAG)
  ((AND (ATOM ATM1) (ATOM ATM2))
    ((NULL FLAG)
      ((if ATM1's print name is lexicographically
          less than ATM2's print name)
        (the position of the first mismatch)) )
    ((if ATM1's print name is lexicographically,
        by ignoring case, less than ATM2's)
      (the position of the first mismatch)) ) )

$ (STRING< 'DOG 'CAT)                    ;DOG ordered after CAT
NIL

$ (STRING< 'CAT 'DOG)                    ;CAT ordered before DOG
0

$ (STRING< 'DOG 'DOGGY)                  ;Differ at 3rd letter
3

$ (STRING<= 'DOG 'DOG)                   ;Have same print name
3
```

STRING-LESSP [*atom*$_1$, *atom*$_2$] Library Macro
STRING-GREATERP [*atom*$_1$, *atom*$_2$] Library Macro
STRING-NOT-GREATERP [*atom*$_1$, *atom*$_2$] Library Macro
STRING-NOT-LESSP [*atom*$_1$, *atom*$_2$] Library Macro
STRING-NOT-EQUAL [*atom*$_1$, *atom*$_2$] Library Macro

These macros perform a lexicographical comparison of their two atomic arguments making no distinction between upper and lower case letters. STRING-LESSP performs a less than test, STRING-GREATERP performs a greater than test, STRING-NOT-GREATERP performs a less than or equal test, STRING-NOT-LESSP performs a greater than or equal test, and STRING-NOT-EQUAL performs a not equal test. If the comparison is true, they return the position of the first character at which the print names fail to match, otherwise they return NIL.

These macros provide no additional functionally and are included in muLISP solely for Common LISP compatibility.

```
(DEFMACRO STRING-LESSP (ATM1 ATM2)
  (LIST 'STRING< ATM1 ATM2 T) )

(DEFMACRO STRING-NOT-GREATERP (ATM1 ATM2)
  (LIST 'STRING<= ATM1 ATM2 T) )

$ (STRING-LESSP "dog" "CAT")        ;DOG ordered after CAT
NIL

$ (STRING-LESSP "cat" "DOG")        ;CAT ordered before DOG
0.

$ (STRING-LESSP "DOG" "doggy")          ;Differ at 3rd letter
3

$ (STRING-NOT-GREATERP "DOG" "DOG")     ;Same print name
3
```

STRING-TRIM [*chars*, *atom*] Function
STRING-LEFT-TRIM [*chars*, *atom*] Function
STRING-RIGHT-TRIM [*chars*, *atom*] Function

If *chars* is a list of characters, (STRING-TRIM *chars atom*) returns a symbol whose print name is *atom*'s print name with the characters in *chars* stripped off both ends. If *chars* is a symbol, STRING-TRIM uses all the characters in *chars* print name instead.

STRING-LEFT-TRIM and STRING-RIGHT-TRIM are similar to STRING-TRIM but they only strip characters off the left and right ends of *atom* respectively.

```
(DEFUN STRING-TRIM (LST ATM)
  (STRING-RIGHT-TRIM LST
                     (STRING-LEFT-TRIM LST ATM)) )

(DEFUN STRING-LEFT-TRIM (LST ATM)
  ((EQ ATM '||) ATM)
  ((MEMBER (CHAR ATM 0) LST)
    (STRING-LEFT-TRIM LST (SUBSTRING ATM 1)) )
  ATM )

$ (STRING-TRIM '| ,.| '|,Joe Doe. |)
|Joe Doe|

$ (STRING-TRIM '(| | |,| |.|) '|,Joe Doe. |)
|Joe Doe|

$ (STRING-LEFT-TRIM '| ,.| '|,Joe Doe. |)
|Joe Doe. |

$ (STRING-RIGHT-TRIM '| ,.| '|,Joe Doe. |)
|,Joe Doe|
```

STRING-UPCASE [*atom*] Function
STRING-DOWNCASE [*atom*] Function
STRING-CAPITALIZE [*atom*] Function

(STRING-UPCASE *atom*) returns the symbol whose print name is the same as *atom*'s print name except that all lowercase characters are converted to uppercase. STRING-DOWNCASE returns the symbol whose print name is the same as *atom*'s print name except that all uppercase characters are converted to lowercase. STRING-CAPITALIZE returns the symbol whose print name is the same as *atom*'s print name except that the first, and only the first, letter of each word is capitalized. If *atom* is not an atom, STRING-UPCASE, STRING-DOWNCASE, and STRING-CAPITALIZE return NIL.

```
(DEFUN STRING-UPCASE (ATM)
  ((ATOM ATM)
    (PACK (MAPCAR 'CHAR-UPCASE (UNPACK ATM))) ))

(DEFUN STRING-DOWNCASE (ATM)
  ((ATOM ATM)
    (PACK (MAPCAR 'CHAR-DOWNCASE (UNPACK ATM))) ))

$ (STRING-UPCASE "Lisp is recursive!")
|LISP IS RECURSIVE!|                    ;All upper case

$ (STRING-DOWNCASE "Lisp is recursive!")
|lisp is recursive!|                    ;All lower case

$ (STRING-CAPITALIZE "Lisp is recursive!")
|Lisp Is Recursive!|                    ;All first letters capitalized
```

FINDSTRING [$atom_1$, $atom_2$, n] Function

(FINDSTRING $atom_1$ $atom_2$ n) returns the position of the first occurrence of $atom_1$'s print name in $atom_2$'s print name, where zero corresponds to the first character of a print name. If n is zero or a positive integer, the search begins with the nth character of $atom_2$. If $atom_1$'s print name is not found, FINDSTRING returns NIL.

```
(DEFUN FINDSTRING (ATM1 ATM2 N)
  ((OR (NOT (ATOM ATM1)) (NOT (ATOM ATM2))) NIL)
  ((PLUSP N)
    ((FINDSTRING ATM1 (SUBLIST ATM2 N))
      (+ N (FINDSTRING ATM1 (SUBLIST ATM2 N))) ) )
  ((if ATM1 occurs as a substring in ATM2)
    (the position of ATM1's first occurrence in
        ATM2) ) )

$ (FINDSTRING 'XYZ 'ABCXYZDEFXYZGHI)
3                              ;Index of first occurrence of XYZ

$ (FINDSTRING 'XYZ 'ABCXYZDEFXYZGHI 4)
9                              ;First occurrence after 4th letter

$ (FINDSTRING 'XYZ 'ABCDEFGHI)
NIL                                          ;XYZ not found
```

PRINT-LENGTH [*atom*] Function

If *atom* is an atom, (PRINT-LENGTH *atom*) returns the number of characters required to print *atom*, taking into account the current value of the control variables *PRINT-ESCAPE* and *PRINT-BASE* (see Section 5.14). Otherwise, PRINT-LENGTH returns NIL.

```
(DEFUN PRINT-LENGTH (ATM)
  ((ATOM ATM)
  (LENGTH (UNPACK ATM)) )

$ (PRINT-LENGTH 'MULISP)                ;Print length of symbol
6

$ (PRINT-LENGTH -100)                   ;Print length of number
4                                               ;in decimal

$ (SETQ *PRINT-BASE* 16)                    ;Print hexadecimal
10

$ (PRINT-LENGTH -100)                   ;Print length of number
3                               ;in hexadecimal (-100 is -64 in hex)

$ (SETQ *PRINT-BASE* 10)                        ;Print decimal
10

$ (PRINT-LENGTH '|Live Aid|)            ;Print length of symbol
10

$ (SETQ *PRINT-ESCAPE* NIL)             ;Print w/o escape chars
NIL.

$ (PRINT-LENGTH '|Live Aid|)                ;New print length
8

$ (SETQ *PRINT-ESCAPE* T)               ;Print with escape chars
T
```

5.11.3 New Symbol Creation

GENSYM [*atom*] — Library Function
GENSYM-PREFIX: G — Library Control variable
GENSYM-COUNT: 0 — Library Control variable

(GENSYM) creates a symbol by concatenating *GENSYM-PREFIX* and *GENSYM-COUNT*, increments *GENSYM-COUNT*, and returns the newly created symbol. If *atom* is a symbol, (GENSYM *atom*) sets *GENSYM-PREFIX* to *atom* before proceeding to create a symbol as described above. If *atom* is zero or a positive integer, (GENSYM *atom*) sets *GENSYM-COUNT* to *atom* before proceeding to create a symbol as described above.

Note that there is no guarantee that the symbol created by GENSYM is actually a new symbol distinct from the other symbols in the current muLISP environment.

```
$ (GENSYM)
G0

$ (GENSYM "GENSYM-")
GENSYM-1

$ (GENSYM 7)
GENSYM-7

$ (GENSYM)
GENSYM-8
```

5.12 NUMERICAL FUNCTIONS

The numerical functions implement the basic mathematical operations for both integer and rational numbers. The user can select either exact or approximate rational arithmetic. For exact rational arithmetic the size of integers, numerators, and denominators is "limited" to roughly 25000 decimal digits! For approximate rational (often called floating slash) arithmetic the precision used can be adjusted to any desired degree of accuracy within this limitation.

If a primitive function requiring integer arguments is called with a noninteger argument, a "Noninteger Argument" error break occurs. If a function requiring numeric (i.e. integer or rational) arguments is called with a nonnumeric argument, a "Nonnumeric Argument" error break occurs. If a numerical function is called with too few arguments, an "Insufficient Arguments" error break occurs. If an attempt is made to divide by zero, a "Zero Divide" error break occurs.

When an error break occurs, a diagnostic message, the errant function call, and the break option prompt are displayed on the console. The break allows the user to determine the cause of the error and how to proceed. Section 5.19 gives more information about breaks.

The control variables *READ-BASE*, *PRINT-BASE*, and *PRINT-POINT* affect how numbers are expressed for input and output (see Sections 5.13 and 5.14). Numerical results in the examples in this section are shown as they would appear with these variables set to their default value.

Although rational numbers can be entered using either decimal notation or slash notation, internally they are stored as the reduced ratio of two integers. Therefore, the call

```
(= -0.75 -3/4 -6/8)
```

is true. Note that when entering rational numbers, there should be no spaces around a dot, slash, or minus sign, and at least one decimal digit should precede a dot.

5.12.1 Precision and Underflow

The functions in this subsection select either exact or approximate rational arithmetic and determine the precision and underflow threshold used for numerical operations.

The rounding algorithm used for the approximate rational arithmetic was inspired by the work of Matula and Kornerup (see D. W. Matula and P. Kornerup, "Finite Precision Rational Arithmetic: Slash Number Systems", IEEE Transactions of Computers, C-34, No. 1, January, 1985, pp. 3-18).

PRECISION [*n*]	Function
EXACT-MODE: NIL	Control variable
RATIONAL	Keyword

If *n* is a positive integer, (PRECISION *n*) sets the precision to *n* and returns the previous precision. If *n* is omitted or is not a positive integer, (PRECISION *n*) returns the current precision without changing it. When muLISP starts-up, the initial precision is 1. The initial precision can be changed by modifying the muLISP Base Page (see Appendix B).

If *EXACT-MODE* is NIL and the current precision is *n*, the numerical functions use approximate rational arithmetic to round results as follows: a) if the exact result is a ratio having a magnitude greater than 65536^n, the result is rounded to the truncated integer quotient of its numerator over its denominator; b) if the exact result is a ratio having a magnitude less than 65536^{-n}, the result is rounded to the reciprocal of the truncated integer quotient of its denominator over its numerator; otherwise c) if the exact result is a ratio requiring a total of more than *n*+1 words of memory to store its numerator and denominator, the result is rounded to one of the two closest rational approximations that require a total of no more than *n*+1 words of memory to store its numerator and denominator. (A word is equivalent to 2 bytes or 16 bits of memory).

In short, a precision of *n* generally provides about 2.5 + 4.5*n* decimal digits of accuracy. Thus, a precision of 1 roughly corresponds to single precision floating point; a precision of 3 roughly corresponds to double precision floating point; and so on.

If *EXACT-MODE* is nonNIL, the rational numerical functions (+, -, *, /, GCD, and LCM) use exact rational arithmetic and do *not* round results. If *EXACT-MODE* is the symbol RATIONAL, the irrational numerical functions (EXP, EXPT, LN, LOG, SQRT, ISQRT, SIN, COS, TAN, ASIN, ACOS, ATAN, and PI) use approximate rational arithmetic to round results as described above. If *EXACT-MODE* is nonNIL and *not* the symbol RATIONAL, the irrational numerical functions generate "Exact Mode" error breaks if called upon to generate irrational numbers.

```
(DEFUN PRECISION (N)
  ((AND (INTEGERP N) (PLUSP N))
    (set precision to N)
    (return previous precision) )
  (return current precision) )

$ 1.00001                                    ;Does round to 1
1

$ (PRECISION 3)                             ;Set double precision
1

$ 1.00001                                    ;Does not round to 1
1.00001

$ (SETQ *EXACT-MODE* T)                      ;Set for exact mode
T

$ (SETQ *PRINT-POINT* 30)                        ;Print 30 digits
30

$ 1.00000000000000000000000000001    ;Does not round to 1
1.00000000000000000000000000001

$ (SETQ *EXACT-MODE* NIL)              ;Set for approximate mode
NIL

$ (PRECISION 1)                             ;Restore single precision
3
```

UNDERFLOW [*n*] Function

If *n* is a positive integer, (UNDERFLOW *n*) sets the value at which underflow occurs to 65536^{-n} and returns the previous underflow setting. If *n* is zero, (UNDERFLOW *n*) prevents underflow from occurring and returns the previous underflow setting. If *n* is omitted or is neither zero nor a positive integer, (UNDERFLOW *n*) returns the current setting of underflow without changing it. When muLISP starts-up, the initial underflow setting is 0. The initial underflow setting can be changed by modifying the muLISP Base Page (see Appendix B).

Sometimes when doing calculations with approximate arithmetic, it is desirable to have numbers smaller in absolute value than a specified number "underflow" to 0. For example, when computing a series approximation of a function, the series can be terminated when the terms of the series underflow to zero.

The following table gives the approximate numerical value at which underflow occurs for various settings of *n*:

n	Underflow	n	Underflow	n	Underflow	n	Underflow
1	$1.53\ 10^{-5}$	4	$5.42\ 10^{-20}$	7	$1.93\ 10^{-34}$	10	$6.84\ 10^{-49}$
2	$2.33\ 10^{-10}$	5	$8.27\ 10^{-25}$	8	$2.94\ 10^{-39}$	11	$1.04\ 10^{-53}$
3	$3.55\ 10^{-15}$	6	$1.26\ 10^{-29}$	9	$4.48\ 10^{-44}$	12	$1.59\ 10^{-58}$

```
(DEFUN UNDERFLOW (N)
  ((ZEROP N)
    (disable underflow feature)
    (return previous underflow setting) )
  ((AND (INTEGERP N) (PLUSP N))
    (set underflow point to 65536^-N)
    (return previous underflow setting) )
  (return current underflow setting) )

$ (UNDERFLOW 7)                    ;Set underflow to 7
0

$ (UNDERFLOW 0)                    ;Disable underflow
7
```

5.12.2 Numerical Recognizers

Since the numerical recognizers are used to identify objects, they do *not* cause "Nonnumeric Argument" error breaks, even if given a symbol or a cons as an argument; they simply return NIL.

ZEROP [*object*] Function

If *object* is the number 0, (ZEROP *object*) returns T, otherwise it returns NIL.

```
(DEFUN ZEROP (OBJ)
  (AND (NUMBERP OBJ) (= OBJ 0)) )

$ (ZEROP 12)
NIL

$ (ZEROP 0)
T

$ (ZEROP 0.0)
T

$ (ZEROP 3.1416)
NIL

$ (ZEROP -7/3)
NIL

$ (ZEROP 'DOG)
NIL
```

PLUSP [*object*] Function
MINUSP [*object*] Function

If *object* is a positive number, (PLUSP *object*) returns T, otherwise it returns NIL. If *object* is a negative number, (MINUSP *object*) returns T, otherwise it returns NIL.

```
(DEFUN PLUSP (OBJ)
  (AND (NUMBERP OBJ) (> OBJ 0)) )

(DEFUN MINUSP (OBJ)
  (AND (NUMBERP OBJ) (< OBJ 0)) )

$ (PLUSP 3.1416)
T

$ (PLUSP -7/3)
NIL

$ (PLUSP 0)
NIL

$ (PLUSP 'DOG)
NIL


$ (MINUSP 3.1416)
NIL

$ (MINUSP -7/3)
T

$ (MINUSP 0)
NIL

$ (MINUSP 'DOG)
NIL
```

ODDP [*object*] Function
EVENP [*object*] Function

If *object* is an odd integer, (ODDP *object*) returns T, otherwise it returns NIL. If *object* is an even integer, (EVENP *object*) returns T, otherwise it returns NIL.

```
(DEFUN ODDP (OBJ)
  (AND (INTEGERP OBJ) (NOT (ZEROP (REM OBJ 2)))) )

(DEFUN EVENP (OBJ)
  (AND (INTEGERP OBJ) (ZEROP (REM OBJ 2))) )

$ (ODDP 12)
NIL

$ (ODDP -41)
T

$ (ODDP 3.1416)
NIL

$ (ODDP 'DOG)
NIL


$ (EVENP 12)
T

$ (EVENP -41)
NIL

$ (EVENP 3.1416)
NIL

$ (EVENP 'DOG)
NIL
```

5.12.3 Numerical Comparators

The numerical comparators are predicates used to compare two or more numbers. They generate "Nonnumeric Argument" error breaks if any argument is *not* a number.

= [n_1, n_2, ..., n_m] Function

If n_1 is equal to n_2, and n_2 is equal to n_3, and ..., and n_{m-1} is equal to n_m, then (= n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (=) and (= *n*) return T. This multi-argument, numerical predicate generates a "Nonnumeric Argument" error break if any argument is not a number.

```
(DEFUN = LST
  ((NULL LST))
  ((NULL (CDR LST)))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((numerically equal (CAR LST) (CADR LST))
      (APPLY '= (CDR LST)) ) )
  ((BREAK (LIST '= (CAR LST) (CADR LST))
          "Nonnumeric Argument")
    (APPLY '= (CDR LST)) ) )

$ (= 5 9)
NIL

$ (= 4 4 -7)
NIL

$ (= 3 3.0)
T

$ (= 0.75 6/8)
T
```

/= $[n_1, n_2, \ldots, n_m]$ Function

If n_1 is *not* equal to n_2, and n_2 is *not* equal to n_3, and ..., and n_{m-1} is *not* equal to n_m, then (/= n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (/=) and (/= n) return T. This multi-argument, numerical predicate generates a "Nonnumeric Argument" error break if any argument is not a number.

Note that if /= returns T, no *adjacent* pair of arguments are equal. However, it can still be the case that two *nonadjacent* arguments are equal.

```
(DEFUN /= LST
  ((NULL LST))
  ((NULL (CDR LST)))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((= (CAR LST) (CADR LST)) NIL)
    (APPLY '/= (CDR LST)) )
  ((BREAK (LIST '/= (CAR LST) (CADR LST))
          "Nonnumeric Argument") NIL )
    (APPLY '/= (CDR LST)) )

$ (/= 5 9)
T

$ (/= 4 4 -7)
NIL

$ (/= 3 3.0)
NIL

$ (/= 0.75 6/8)
NIL

$ (/= 6 -2 6)                    ;No adjacent pairs are equal
T
```

< [n_1, n_2, ..., n_m] — Function

If $n_1 < n_2$, and $n_2 < n_3$, and .... and $n_{m-1} < n_m$, then (< n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (<) and (< *n*) return T. This multi-argument, numerical predicate generates a "Nonnumeric Argument" error break if any argument is not a number.

The predicate < is commonly used to determine if a number lies within a given *interval* by calling it with three arguments; the number and the interval's end-points. For example, the following tests respectively determine if the value of CHAR is a numeral, an upper-case letter, or a lower-case letter:

```
(< 47 (ASCII CHAR) 58)
(< 64 (ASCII CHAR) 91)
(< 96 (ASCII CHAR) 123)
```

```
(DEFUN < LST
  ((NULL LST))
  ((NULL (CDR LST)))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
   ((less than (CAR LST) (CADR LST))
     (APPLY '< (CDR LST)) ) )
  ((BREAK (LIST '< (CAR LST) (CADR LST))
          "Nonnumeric Argument")
    (APPLY '< (CDR LST)) ) )
```

```
$ (< 5 9)
T

$ (< 4 -7)
NIL

$ (< 3 3.0)
NIL

$ (< 3/5 2/3)
T
```

> [n_1, n_2, ..., n_m] — Function

If $n_1 > n_2$, and $n_2 > n_3$, and ..., and $n_{m-1} > n_m$, then (> n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (>) and (> n) return T. This multi-argument, numerical predicate generates a "Nonnumeric Argument" error break if any argument is not a number.

```
(DEFUN > LST
  ((NULL LST))
  ((NULL (CDR LST)))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((greater than (CAR LST) (CADR LST))
      (APPLY '> (CDR LST)) ) )
  ((BREAK (LIST '> (CAR LST) (CADR LST))
          "Nonnumeric Argument")
    (APPLY '> (CDR LST)) ) )

$ (> 5 9)
NIL

$ (> 4 -7)
T

$ (> 3 3.0)
NIL

$ (> 3/5 2/3)
NIL
```

<= [n_1, n_2, ... n_m] Function

If $n_1 \leq n_2$, and $n_2 \leq n_3$, and ..., and $n_{m-1} \leq n_m$, then (<= n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (<=) and (<= n) return T. This multi-argument, numerical predicate generates a "Nonnumeric Argument" error break if any argument is not a number.

The predicate <= is commonly used to determine if a number lies in a given *interval* by calling it with three arguments; the number and the interval's end-points. For example, the following tests respectively determine if the value of CHAR is a numeral, an upper-case letter, or a lower-case letter:

```
(<= 48 (ASCII CHAR) 57)
(<= 65 (ASCII CHAR) 90)
(<= 97 (ASCII CHAR) 122)
```

```
(DEFUN <= LST
  ((NULL LST))
  ((NULL (CDR LST)))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((> (CAR LST) (CADR LST)) NIL)
    (APPLY '<= (CDR LST)) )
  ((BREAK (LIST '<= (CAR LST) (CADR LST))
          "Nonnumeric Argument")
    (APPLY '<= (CDR LST)) ) )

$ (<= 5 9)
T

$ (<= 4 -7)
NIL

$ (<= 3 3.0)
T

$ (<= 3/5 2/3)
T
```

>= [n_1, n_2, ..., n_m] Function

If $n_1 \geq n_2$, and $n_2 \geq n_3$, and .... $n_{m-1} \geq n_m$, then (>= n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (>=) and (>= n) return T. This multi-argument, numerical predicate generates a "Nonnumeric Argument" error break if any argument is not a number.

```
(DEFUN >= LST
  ((NULL LST))
  ((NULL (CDR LST)))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((< (CAR LST) (CADR LST)) NIL)
    (APPLY '>= (CDR LST)) )
  ((BREAK (LIST '>= (CAR LST) (CADR LST))
          "Nonnumeric Argument")
    (APPLY '>= (CDR LST)) ) )

$ (>= 5 9)
NIL

$ (>= 4 -7)
T

$ (>= 3 3.0)
T

$ (>= 3/5 2/3)
NIL
```

MAX $[n_1, n_2, ..., n_m]$ Function
MIN $[n_1, n_2, ..., n_m]$ Function

MAX returns its greatest argument (i.e. the argument furthest to the right on the real number line). MIN returns its least argument (i.e. the argument furthest to the left on the real number line). If MAX or MIN is called with no arguments, an "Insufficient Arguments" error break occurs.

```
(DEFUN MAX LST
  ((NULL LST)
    (BREAK (LIST 'MAX) "Insufficient Arguments") )
  ((NULL (CDR LST))
    (CAR LST) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((< (CAR LST) (CADR LST))
      (APPLY 'MAX (CONS (CADR LST) (CDDR LST))) )
    (APPLY 'MAX (CONS (CAR LST) (CDDR LST))) )
  (APPLY 'MAX (CONS
            (BREAK (LIST 'MAX (CAR LST) (CADR LST))
                   "Nonnumeric Argument")
            (CDDR LST))) )

$ (MAX 5 -7 4)
5

$ (MAX 2/3 0.6)
0.6666666

$ (MAX -4)
-4

$ (MIN 5 -7 4)
-7

$ (MIN 2/3 0.6)
0.6
```

5.12.4 Arithmetic Functions

+ [n_1, n_2, ..., n_m] Function

(+ n_1 n_2 ... n_m) returns the sum of the numbers n_1 through n_m. If called with no arguments, + returns 0, which is the identity for this operation.

```
(DEFUN + LST
  ((NULL LST) 0)
  ((NULL (CDR LST))
    (CAR LST) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    (APPLY '+ (CONS ((CAR LST) plus (CADR LST))
                    (CDDR LST) ) ) )
  (APPLY '+ (CONS (+TRAP (CAR LST) (CADR LST))
                  (CDDR LST) ) ) )

$ (+ 2 3 4)
9

$ (+ -5 3)
-2

$ (+ 1/6 0.7)
0.8666666

$ (+ 0.25)
0.25
```

- [n_1, n_2, ..., n_m] — Function

(- n_1 n_2 ... n_m) returns the difference between n_1 and the sum of n_2 through n_m. (- n) returns minus n. If - is called with no arguments, an "Insufficient Arguments" error break occurs.

```
(DEFUN - LST
  ((NULL LST)
    (BREAK (LIST '-) "Insufficient Arguments") )
  ((NULL (CDR LST))
    ((NUMBERP (CAR LST))
      (minus (CAR LST)) )
    (-TRAP (CAR LST)) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((NULL (CDDR LST))
      ((CAR LST) minus (CADR LST)) )
    (APPLY '- (CONS ((CAR LST) minus (CADR LST))
                    (CDDR LST) ) ) )
  (APPLY '- (CONS (-TRAP (CAR LST) (CADR LST))
                  (CDDR LST) ) ) )

$ (- 12 5)
7

$ (- 12 5 -3)
10

$ (- 17/9 1.5)
0.3888888

$ (- 8)
-8
```

* [n_1, n_2, ..., n_m] — Function

(* n_1 n_2 ... n_m) returns the product of the numbers n_1 through n_m. If called with no arguments, * returns 1, which is the identity for this operation.

```
(DEFUN * LST
  ((NULL LST) 1)
  ((NULL (CDR LST))
    (CAR LST) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    (APPLY '* (CONS ((CAR LST) times (CADR LST))
                    (CDDR LST) ) ) )
  (APPLY '* (CONS (*TRAP (CAR LST) (CADR LST))
                  (CDDR LST) ) ) )

$ (* 3 4 5)
60

$ (* 5/6 -0.7)
-0.5833333

$ (*)
1

$ (* 2/5)
0.4
```

/ [n_1, n_2, ... n_m]

Function

(/ n_1 n_2 ... n_m) returns the result of n_1 divided by the product of n_2 through n_m. If called with a single argument, / returns the reciprocal of that argument. If / is called with no arguments, an "Insufficient Arguments" error break occurs. If any argument other than the first is zero, a "Zero Divide" error break occurs.

```
(DEFUN / LST
  ((NULL LST)
    (BREAK (LIST '/) "Insufficient Arguments") )
  ((NULL (CDR LST))
    ((NUMBERP (CAR LST))
      ((ZEROP (CAR LST))  (/TRAP (CAR LST)) )
      (reciprocal of (CAR LST)) )
    (/TRAP (CAR LST)) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((NULL (CDDR LST))
      ((ZEROP (CADR LST))
        (/TRAP (CAR LST) (CADR LST)) )
      (quotient (CAR LST) (CADR LST)) )
    ((ZEROP (CADR LST))
      (APPLY '/ (CONS (/TRAP (CAR LST) (CADR LST))
                      (CDDR LST) ) ) )
    (APPLY '/ (CONS (quotient (CAR LST)(CADR LST))
                    (CDDR LST) ) ) )
  (APPLY '/ (CONS (/TRAP (CAR LST) (CADR LST))
                  (CDDR LST) ) ) )
```

```
$ (/ 12 8)
1.5

$ (/ 12 5 -3)
-0.8

$ (/ -4.7 1.3)
-3.6153846

$ (/ 5)
0.2
```

ADD1 [*n*] Function
SUB1 [*n*] Function

(ADD1 *n*) returns (+ *n* 1). (SUB1 *n*) returns (- *n* 1). If ADD1 or SUB1 is called with no arguments, an "Insufficient Arguments" error break occurs.

```
(DEFUN ADD1 (N)
  ((NUMBERP N)
    (+ N 1) )
  (BREAK (LIST 'ADD1 N) "Nonnumeric Argument") )

(DEFUN SUB1 (N)
  ((NUMBERP N)
    (- N 1) )
  (BREAK (LIST 'SUB1 N) "Nonnumeric Argument") )

$ (ADD1 3)
4

$ (ADD1 2/3)
1.6666666

$ (ADD1 -3)
-2

$ (SUB1 3)
2

$ (SUB1 0.25)
-0.75

$ (SUB1 -3)
-4
```

INCQ [*symbol*, *n*] Special form
DECQ [*symbol*, *n*] Special form

If *n* is a number, (INCQ *symbol n*) adds *n* to the value of *symbol*, sets *symbol* to the result, and returns the result. If *n* is omitted, *symbol* is incremented by 1.

If *n* is a number, (DECQ *symbol n*) subtracts *n* from the value of *symbol*, sets *symbol* to the result, and returns the result. If *n* is omitted, *symbol* is decremented by 1.

If *symbol* is not a symbol, INCQ and DECQ cause a "Nonsymbolic Argument" error break.

```
(DEFMACRO INCQ (SYM N)
  ((NULL N)
    (LIST 'SETQ SYM (LIST 'ADD1 SYM)) )
  (LIST 'SETQ SYM (LIST '+ SYM N)) )

(DEFMACRO DECQ (SYM N)
  ((NULL N)
    (LIST 'SETQ SYM (LIST 'SUB1 SYM)) )
  (LIST 'SETQ SYM (LIST '- SYM N)) )

$ (SETQ CTR 5)                          ;Set counter to 5
5

$ (INCQ CTR)                            ;Increment counter
6

$ (INCQ CTR 2)                          ;Add 2 to counter
8

$ (DECQ CTR 5)                          ;Subtract 5 from counter
3

$ CTR                                   ;Value of counter
3
```

INCF [*place*, *n*] Library Macro
DECF [*place*, *n*] Library Macro

If *place* is a valid place form and *n* is a number, (INCF *place n*) adds *n* to the value in *place*, stores the result in *place*, and returns the result. If *n* is omitted, *place* is incremented by 1.

If *place* is a valid place form and *n* is a number, (DECF *place n*) subtracts *n* from the value in *place*, stores the result in *place*, and returns the result. If *n* is omitted, *place* is decremented by 1.

If *place* is not a valid place form (see SETF in Section 5.7), INCF and DECF generate an "Invalid Place Form" error break.

```
(DEFMACRO INCF (PLACE N)
  ((NULL N)
    (MAKE-SET-FORM PLACE (LIST 'ADD1 PLACE)) )
  (MAKE-SET-FORM PLACE (LIST '+ PLACE N)) )

(DEFMACRO DECF (PLACE N)
  ((NULL N)
    (MAKE-SET-FORM PLACE (LIST 'SUB1 PLACE)) )
  (MAKE-SET-FORM PLACE (LIST '- PLACE N)) )

$ (SETQ LST '(5 2 3))                          ;Initialize LST
(5 2 3)

$ (INCF (CAR LST))                          ;Increment car of LST
6

$ (INCF (NTH 1 LST) 2)                   ;Add 2 to second of LST
4

$ (DECF (CADR LST) 5)              ;Subtract 5 from second of LST
-1

$ LST                                                ;Value of LST
(6 -1 3)
```

GCD [n_1, n_2, ..., n_m] Function

(GCD n_1 n_2 ... n_m) returns the greatest common divisor of the numbers n_1 through n_m. (GCD n) returns the absolute value of n. If called with no arguments, GCD returns 0, which is the identity for this operation.

The GCD of two integers is the largest non-negative integer that evenly divides both integers. The GCD of two rational numbers is the GCD of the numerators divided by the LCM of the denominators.

```
(DEFUN GCD LST
  ((NULL LST) 0)
  ((NULL (CDR LST))
    ((NUMBERP (CAR LST))
      (ABS (CAR LST)) )
    (GCDTRAP (CAR LST)) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    (APPLY 'GCD (CONS
           (/ (Integer-GCD (NUMERATOR (CAR LST))
                           (NUMERATOR (CADR LST)))
              (Integer-LCM (DENOMINATOR (CAR LST))
                           (DENOMINATOR (CADR LST))))
            (CDDR LST) ) ) )
  (APPLY 'GCD (CONS (GCDTRAP (CAR LST) (CADR LST))
                    (CDDR LST) ) ) )

(DEFUN Integer-GCD (N M)
  ((ZEROP M)
    (ABS N) )
  (Integer-GCD M (REM N M)) )

(DEFUN Integer-LCM (N M)
  (/ (* N M) (Integer-GCD N M)) )
```

```
$ (GCD 12 -16 20)
4

$ (GCD 14/15 8/9)
0.0444444

$ (GCD 0 5)
5

$ (GCD 0 0)
0

$ (GCD)
0

$ (GCD -7)
7
```

LCM [n_1, n_2, ..., n_m] Function

(LCM n_1 n_2 ... n_m) returns the least common multiple of the numbers n_1 through n_m. (LCM n) returns the absolute value of n. If LCM is called with no arguments, an "Insufficient Arguments" error break occurs since LCM has no finite identity.

The LCM of two integers is the least non-negative integer that is a multiple of both integers. In general, the LCM of the two numbers n and m, either integer or rational, can be defined in terms of GCD as

(/ (ABS (* n m)) (GCD n m)).

```
(DEFUN LCM LST
  ((NULL LST)
    (BREAK (LIST 'LCM) "Insufficient Arguments") )
  ((NULL (CDR LST))
    ((NUMBERP (CAR LST))
      (ABS (CAR LST)) )
    (LCMTRAP (CAR LST)) )
  ((OR (ZEROP (CAR LST)) (ZEROP (CADR LST))) 0)
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    (APPLY 'LCM (CONS
                  (/ (ABS (* (CAR LST) (CADR LST)))
                     (GCD (CAR LST) (CADR LST)))
                  (CDDR LST) ) ) )
  (APPLY 'LCM (CONS (LCMTRAP (CAR LST) (CADR LST))
                    (CDDR LST) ) ) )

$ (LCM 12 -16 20)
240

$ (LCM 14/15 8/9)
18.6666666                    ;This result is = to 56/3

$ (LCM 0 5)
0
```

5.12.5 Irrational and Transcendental Functions

The functions described in this subsection are defined in the library file IRRATNAL.LSP that accompanies muLISP. Most of the functions generate an "Exact Mode" error break if the predicate EXACT-MODEP returns T (i.e. if the value of the control variable *EXACT-MODE* is neither NIL nor the symbol RATIONAL (see Subsection 5.12.1)).

EXP [*n*] Library Function

If *n* is a number, (EXP *n*) returns e^n. *e* is the base of the natural logarithms (i.e. 2.71828 ...). The result is as accurate as the current precision setting will allow. If exact mode arithmetic is in effect, an "Exact Mode" error break occurs. EXP uses and updates if necessary the variable *E* to store *e* accurate to the current precision.

```
(DEFUN EXP (N)
  ((NUMBERP N)
    ((ZEROP N) 1)
    ((EXACT-MODEP)
      (BREAK (LIST 'EXP N) "Exact Mode") )
    (EXP-CALC N) )
  (BREAK (LIST 'EXP N) "Nonnumeric Argument") )

$ (EXP 1)                                   ;Compute e to 7 digits
2.7182818

$ (PROGN (PRECISION 10) (SETQ *PRINT-POINT* 50))
50                                ;Set precision to 10 & print 50 digits

$ (EXP 1)                                   ;Compute e to about 45 digits
2.7182818284590452353602874713526624977572470937000
07

$ (PROGN (PRECISION 1) (SETQ *PRINT-POINT* 7))
7                                 ;Restore precision to 1 & print 7 digits
```

EXPT [*n*, *m*] Library Function

If *n* and *m* are numbers, (EXPT *n m*) returns n^m. If *m* is a ratio, the result is as accurate as the current precision setting will allow. (EXPT *n* 0) returns 1, even if *n* is 0. If *n* is negative and the denominator of *m* is even, an "Invalid Argument" error break occurs.

```
(DEFUN EXPT (N M
    ANS)
  ((AND (NUMBERP N) (NUMBERP M))
    ((= N 1) 1)
    ((ZEROP M) 1)
    ((= M 1) N)
    ((MINUSP M)
      (/ 1 (EXPT N (- M))) )
    ((ZEROP N) 0)
    ((INTEGERP M)
      (INTEGER-POWER N M) )
    ((MINUSP N)
      ((AND (ODDP (DENOMINATOR M))
            (SETQ ANS (FRACTIONAL-POWER (- N) M)))
        ((EVENP (NUMERATOR M)) ANS)
        (- ANS) )
      (BREAK (LIST 'EXPT N M) "Invalid Argument"))
    ((FRACTIONAL-POWER N M))
    (BREAK (LIST 'EXPT N M) "Exact Mode") )
  (BREAK (LIST 'EXPT N M) "Nonnumeric Argument") )

$ (EXPT 2 3)                                    ;2 cubed
8

$ (EXPT 3 1/3)                                  ;The cube root of 3
1.4422495

$ (EXPT -8 1/3)                                 ;The cube root of -8
-2

$ (EXPT 5 -3)                                   ;5 to the -3
0.008
```

LN [*n*] Library Function
LOG [*n*, *base*] Library Function

If *n* is a positive number, (LN *n*) returns the natural logarithm of *n*. If *n* and *base* are positive numbers, (LOG *n base*) returns the logarithm of *n* in the base *base*. *base* defaults to *e*, the base natural logarithms (i.e. 2.71828...). The result is as accurate as the current precision setting will allow. If *n* or *base* is *not* positive, an "Invalid Argument" error break occurs. LN and LOG use and update if necessary the variable *LN2* to store ln 2, accurate to the current precision.

```
(DEFUN LN (N)
  ((NUMBERP N)
    ((PLUSP N)
      ((= N 1) 0)
      ((EXACT-MODEP)
        (BREAK (LIST 'LN N) "Exact Mode") )
      (LN-CALC N) )
    (BREAK (LIST 'LN N) "Invalid Argument") )
  (BREAK (LIST 'LN N) "Nonnumeric Argument") )

(DEFUN LOG (N M)
  (SETQ N (LN N))
  ((NULL M) N)
  ((ZEROP M) 0)
  (/ N (LN M)) )

$ (LN 2)                              ;The natural log of 2
0.6931471

$ (LOG 2 10)                          ;The log of 2 to the base 10
0.3010299

$ (EXPT 10 *)                         ;Raise 10 to the previous answer
2
```

SQRT [*n*] Library Function
ISQRT [*n*] Library Function

If *n* is a non-negative number, (SQRT *n*) returns the principal square root of *n*. The result is as accurate as the current precision setting will allow. If *n* is negative, an "Invalid Argument" error break occurs.

If *n* is a non-negative integer, (ISQRT *n*) returns the greatest integer less than or equal to the positive square root of *n*. ISQRT uses Newton's method to compute the square root. If *n* is negative, an "Invalid Argument" error break occurs.

```
(DEFUN SQRT (N)
  (EXPT N 1/2) )

(DEFUN ISQRT (N)
  ((INTEGERP N)
    ((>= N 0)
      ((ZEROP N) 0)
      (ISQRT-CALC N) )
    (BREAK (LIST 'ISQRT N) "Invalid Argument") )
  (BREAK (LIST 'ISQRT N) "Noninteger Argument") )

$ (SQRT 5)                          ;The square root of 5
2.2360677

$ (* * *)                           ;Square the previous answer
5

$ (ISQRT 5)                         ;The integer square root of 5
2

$ (ISQRT 1000000)                   ;The integer square root of 1000000
1000
```

SIN [*n*] Library Function
COS [*n*] Library Function
TAN [*n*] Library Function

If *n* is a number, (SIN *n*) returns the sine of *n* radians, (COS *n*) returns the cosine of *n* radians, and (TAN *n*) returns the tangent of *n* radians. The result is as accurate as the current precision setting will allow. If exact mode arithmetic is in effect, an "Exact Mode" error break occurs.

```
(DEFUN SIN (N)
  ((NUMBERP N)
    ((ZEROP N) 0)
    ((EXACT-MODEP)
      (BREAK (LIST 'SIN N) "Exact Mode") )
    (SETQ N (/ N (/ (PI) 4)))
    (SINCOS-CALC (REM N) (TRUNCATE N)) )
  (BREAK (LIST 'SIN N) "Nonnumeric Argument") )

(DEFUN TAN (N)
  (/ (SIN N) (COS N)) )

$ (SIN 0.37)                              ;Sine of 0.37 radians
0.3616154

$ (COS (/ (PI) 4))                        ;Cosine of pi/4 radians (45°)
0.7071068

$ (TAN (/ (PI) 4))                        ;Tangent of pi/4 radians (45°)
1                    ;Result exact, thanks to floating slash arithmetic!
```

ASIN [*n*] — Library Function
ACOS [*n*] — Library Function
ATAN [*n*, *m*] — Library Function

If *n* is a number between -1 and 1 inclusive, (ASIN *n*) returns in radians the arc sine of *n* and (ACOS *n*) returns in radians the arc cosine of *n*.

(ATAN *n m*) returns in radians the arc tangent (i.e. the angle corresponding to the vector whose opposite component is *n* and whose adjacent component is *m*. *m* defaults to 1.

For all numbers *n* and *m*:

$$\begin{array}{rcrcl} -pi/2 & \leq & (\text{ASIN } n) & \leq & pi/2 \\ 0 & \leq & (\text{ACOS } n) & \leq & pi \\ -pi & < & (\text{ATAN } n\ m) & \leq & pi \end{array}$$

All results are as accurate as the current precision setting will allow. If exact mode arithmetic is in effect, an "Exact Mode" error break occurs.

```
(DEFUN ASIN (N)
  ((NUMBERP N)
    ((EXACT-MODEP)
      (BREAK (LIST 'ASIN N) "Exact Mode") )
    ((MINUSP N)
      (- (ASIN (- N))) )
    ((= N 1)
      (/ (PI) 2) )
    ((< N 1)
      (ATAN (/ N (SQRT (- 1 (* N N))))) )
    (BREAK (LIST 'ASIN N) "Invalid Argument") )
  (BREAK (LIST 'ASIN N) "Nonnumeric Argument") )
```

```
$ (ACOS 0)                          ;The angle whose cosine is 0
1.5707964

$ (ATAN 2 3)                        ;The angle whose opposite is 2
0.5880026                               ;and whose adjacent is 3
```

PI [] Library Function

(PI) returns *pi*. The result is as accurate as the current precision setting will allow. If exact mode arithmetic is in effect, an "Exact Mode" error break occurs. PI uses and updates if necessary the variable *PI* to store *pi* accurate to the current precision.

In the following examples the inaccurate digits are high-lighted by underlining.

```
(DEFUN PI ()
  ((EXACT-MODEP)
    (BREAK (CONS 'PI) "Exact Mode") )
  (PI-CALC) )

$ (PI)                                   ;Compute pi to 7 digits
3.1415926

$ (PROGN (PRECISION 18) (SETQ *PRINT-POINT* 98))
98                                ;Set precision to 18 & print 98 digits

$ (PI)                                   ;Compute pi to about 80 digits
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803520847034814

$ (PRECISION 20)                                  ;Set precision to 20
18

$ (PI)                                   ;Compute pi to about 90 digits
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211732

$ (PROGN (PRECISION 1) (SETQ *PRINT-POINT* 7))
7                                 ;Restore precision to 1 & print 7 digits
```

5.12.6 Numerical Selectors

ABS [*n*] Function

If *n* is a number, (ABS *n*) returns the absolute value of *n*.

```
(DEFUN ABS (N)
  ((NUMBERP N)
    ((MINUSP N)
      (- N) )
    N )
  (ABSTRAP N) )

$ (ABS 4/6)
0.6666666

$ (ABS -3)
3

$ (ABS 0)
0
```

:IGNUM [*n*] — Function

:f *n* is a number, (SIGNUM *n*) returns 1, -1, or 0 depending on whether : is positive, negative, or zero. If *n* is not a number, SIGNUM generates : "Nonnumeric Argument" error break.

```
(DEFUN SIGNUM (N)
  ((NUMBERP N)
    ((ZEROP N) 0)
    ((PLUSP N) 1)
    -1 )
  (BREAK (LIST 'SIGNUM N) "Nonnumeric Argument") )

$ (SIGNUM 7/2)
1

$ (SIGNUM 0)
0

$ (SIGNUM -0.2)
-1
```

NUMERATOR [*n*] Function
DENOMINATOR [*n*] Function

If *n* is a ratio, (NUMERATOR *n*) returns the numerator of *n*. If *n* is an integer, (NUMERATOR *n*) returns *n*. The numerator is always an integer.

If *n* is a ratio, (DENOMINATOR *n*) returns the denominator of *n*. If *n* is an integer, (DENOMINATOR *n*) returns 1. The denominator is always a positive integer.

```
(DEFUN NUMERATOR (N)
  ((NUMBERP N)
    ((INTEGERP N) N)
    (the numerator of N) )
  (NUMTRAP N) )

(DEFUN DENOMINATOR (N)
  ((NUMBERP N)
    ((INTEGERP N) 1)
    (the denominator of N) )
  (DENTRAP N) )

$ (NUMERATOR 10/8)
5

$ (NUMERATOR 0.75)
3

$ (NUMERATOR -8)
-8

$ (DENOMINATOR 10/8)
4

$ (DENOMINATOR 0.75)
4
```

FLOOR [*n*, *m*] Function

If *n* is an integer, (FLOOR *n*) returns *n*. If *n* is a ratio, FLOOR returns the largest integer less than *n*. In other words, FLOOR truncates *n* toward minus infinity.

If *n* and *m* are numbers, (FLOOR *n m*) truncates toward minus infinity the quotient of *n* divided by *m*. (FLOOR *n m*) is equivalent to (FLOOR (/ *n m*)), but more efficient. If *m* (i.e. the divisor) is zero, a "Zero Divide" error break occurs.

```
(DEFUN FLOOR (N M)
  ((NUMBERP M)
    ((ZEROP M)
      (BREAK (LIST 'FLOOR N M) "Zero Divide") )
    (FLOOR (/ N M)) )
  ((NUMBERP N)
    ((INTEGERP N) N)
    (the largest integer less than N) )
  (BREAK (LIST 'FLOOR N M) "Nonnumeric Argument"))

$ (FLOOR 7.3)
7

$ (FLOOR -3.5)
-4

$ (FLOOR 8 3)
2

$ (FLOOR -15/4 2)
-2
```

CEILING [*n*, *m*] Function

If *n* is an integer, (CEILING *n*) returns *n*. If *n* is a ratio, CEILING returns the smallest integer greater than *n*. In other words, CEILING truncates *n* toward plus infinity.

If *n* and *m* are numbers, (CEILING *n m*) truncates toward plus infinity the quotient of *n* divided by *m*. (CEILING *n m*) is equivalent to (CEILING (/ *n m*)), but more efficient. If *m* (i.e. the divisor) is zero, a "Zero Divide" error break occurs.

```
(DEFUN CEILING (N M)
  ((NUMBERP M)
    ((ZEROP M)
      (BREAK (LIST 'CEILING N M) "Zero Divide") )
    (CEILING (/ N M)) )
  ((NUMBERP N)
    ((INTEGERP N) N)
    (the smallest integer greater than N) )
  (BREAK (LIST 'CEILING N M)
         "Nonnumeric Argument") )

$ (CEILING 7.3)
8

$ (CEILING -3.5)
-3

$ (CEILING 8 3)
3

$ (CEILING -15/4 2)
-1
```

TRUNCATE [*n*, *m*] Function

If *n* is an integer, (TRUNCATE *n*) returns *n*. If *n* is a positive ratio, TRUNCATE returns the largest integer less than *n*. If *n* is a negative ratio, TRUNCATE returns the smallest integer greater than *n*. In other words, TRUNCATE truncates *n* toward zero.

If *n* and *m* are numbers, (TRUNCATE *n m*) truncates toward zero the quotient of *n* divided by *m*. (TRUNCATE *n m*) is equivalent to (TRUNCATE (/ *n m*)), but more efficient. If *m* (i.e. the divisor) is zero, a "Zero Divide" error break occurs.

```
(DEFUN TRUNCATE (N M)
  ((NUMBERP M)
    ((ZEROP M)
      (BREAK (LIST 'TRUNCATE N M) "Zero Divide") )
    (TRUNCATE (/ N M)) )
  ((NUMBERP N)
    ((INTEGERP N) N)
    ((PLUSP N)
      (the largest integer less than N) )
    (the smallest integer greater than N) )
  (BREAK (LIST 'TRUNCATE N M)
         "Nonnumeric Argument") )

$ (TRUNCATE 7.3)
7

$ (TRUNCATE -3.5)
-3

$ (TRUNCATE 8 3)
2

$ (TRUNCATE -15/4 2)
-1
```

ROUND [*n*, *m*] Function

If *n* is an integer, (ROUND *n*) returns *n*. If *n* is a ratio, ROUND returns the integer closest in value to *n*. If *n* is equidistant between two integers (i.e. 2**n* is an integer), ROUND returns the one that is divisible by two.

If *n* and *m* are numbers, (ROUND *n m*) rounds the quotient of *n* divided by *m*. The call (ROUND *n m*) is equivalent to (ROUND (/ *n m*)), but more efficient. If *m* (i.e. the divisor) is zero, a "Zero Divide" error break occurs.

```
(DEFUN ROUND (N M)
  ((NUMBERP M)
    ((ZEROP M)
      (BREAK (LIST 'ROUND N M) "Zero Divide") )
    (ROUND (/ N M)) )
  ((NUMBERP N)
    ((INTEGERP N) N)
    ((INTEGERP (* 2 N))
      ((EVENP (TRUNCATE N))
        (TRUNCATE N) )
      (ADD1 (TRUNCATE N)) )
    (the integer closest to N) )
  (BREAK (LIST 'ROUND N M) "Nonnumeric Argument"))

$ (ROUND 7.3)
7

$ (ROUND -3.5)
-4

$ (ROUND -4.5)
-4

$ (ROUND 8 3)
3

$ (ROUND -15/4 2)
-2
```

MOD [*n*, *m*] Function
REM [*n*, *m*] Function

If *n* and *m* are numbers, (MOD *n m*) returns the modulus of *n* divided by *m*. MOD is defined so that FLOOR and MOD preserve the fundamental relation between quotients and remainders:

$$n = m * (\text{FLOOR } n\ m) + (\text{MOD } n\ m)$$

If *n* and *m* are numbers, (REM *n m*) returns the remainder of *n* divided by *m*. REM is defined so that TRUNCATE and REM preserve the fundamental relation between quotients and remainders:

$$n = m * (\text{TRUNCATE } n\ m) + (\text{REM } n\ m)$$

For both MOD and REM, if *m* (i.e. the divisor) is zero, a "Zero Divide" error break occurs. *m* defaults to 1.

```
(DEFUN MOD (N M)
  ((AND (NUMBERP N) (NUMBERP M))
    ((ZEROP M)
      (BREAK (LIST 'MOD N M) "Zero Divide") )
    (- N (* M (FLOOR N M))) )
  (BREAK (LIST 'MOD N M) "Nonnumeric Argument") )
```

```
$ (MOD 7 3)
1

$ (MOD -7 3)
2

$ (MOD 7 -3)
-2

$ (MOD -7 -3)
-1
```

```
$ (REM 7 3)
1

$ (REM -7 3)
-1

$ (REM 7 -3)
1

$ (REM -7 -3)
-1
```

DIVIDE [*n*, *m*] Function

If *n* is a number, (DIVIDE *n*) returns a cons whose car is (TRUNCATE *n*) and whose cdr is (REM *n*).

If *n* and *m* are numbers, (DIVIDE *n m*) returns a cons whose car is (TRUNCATE *n m*) and whose cdr is (REM *n m*). If *m* (i.e. the divisor) is zero, a "Zero Divide" error break occurs.

When both a truncated quotient and remainder are required, using the DIVIDE function is more efficient than computing them separately.

```
(DEFUN DIVIDE (N M)
  ((AND (NUMBERP N) (NUMBERP M))
    ((ZEROP M)
      (BREAK (LIST 'DIVIDE N M) "Zero Divide") )
    (CONS (TRUNCATE N M) (REM N M)) )
  (BREAK (LIST 'DIVIDE N M)
         "Nonnumeric Argument"))

$ (DIVIDE 7 3)
(2 . 1)

$ (DIVIDE -7 3)
(-2 . -1)

$ (DIVIDE 7 -3)
(-2 . 1)

$ (DIVIDE -7 -3)
(2 . -1)
```

5.12.7 Bit-wise Logical Functions

The bit-wise logical functions in this subsection operate over the integers. If called with noninteger arguments, a "Noninteger Argument" error break occurs.

LOGNOT [*n*]	Function
LOGTEST [*n*, *m*]	Library Function
LOGBITP [*index*, *n*]	Library Function

(LOGNOT *n*) returns the bit-wise logical *not* of *n* (i.e. the integer whose bits are the complement of the bits in *n*). Since LOGNOT operates as if negative integers are stored in "half-infinite" two's-complement notation, the result is always of the opposite sign as *n*.

If any 1 bits in *n* are also 1 bits in *m*, (LOGTEST *n m*) returns T, otherwise it returns NIL.

If the bit in *n* whose weight is 2^{index} is 1, (LOGBITP *index n*) returns T, otherwise it returns NIL.

```
$ (LOGNOT 9)  -                          ;Complement 9
-10

$ (LOGNOT -1)                            ;Complement -1
0

$ (LOGTEST 4 13)                         ;Test the 2nd bit in 13
T

$ (LOGBITP 2 13)                         ;Test the 2nd bit in 13
T
```

LOGIOR [n_1, n_2, ..., n_m] Function
LOGXOR [n_1, n_2, ..., n_m] Function
LOGAND [n_1, n_2, ..., n_m] Function
LOGEQV [n_1, n_2, ..., n_m] Library Function

These functions perform logical operations on the bits of their integer arguments. They operate as if integers are stored in "half-infinite" two's-complement notation.

LOGIOR returns the bit-wise logical *inclusive or* of its arguments. LOGXOR returns the bit-wise logical *exclusive or* of its arguments. LOGAND returns the bit-wise logical *and* of its arguments. LOGEQV returns the bit-wise logical *equivalence* of its arguments. If called without arguments, these functions return the identity for their respective operation (i.e. (LOGIOR) and (LOGXOR) return 0, (LOGAND) and (LOGEQV) return -1).

```
(DEFUN LOGIOR LST
  (REDUCE '(LAMBDA (INT1 INT2)
      ((AND (INTEGERP INT1) (INTEGERP INT2))
        (the logical OR of INT1 and INT2) )
      (BREAK (LIST 'LOGIOR INT1 INT2)
             "Noninteger Argument") )
    LST 0) )

$ (LOGIOR 11 6)
15

$ (LOGXOR -6 6)
-4

$ (LOGAND 11 6)
2

$ (LOGEQV 11 6)
-14
```

SHIFT [*n*, *count*] Function
ASH [*n*, *count*] Library Function

If *n* is an integer and if *count* is a positive integer, (SHIFT *n count*) returns the result of shifting *n* left by *count* bits. If *count* is a negative integer, *n* is shifted right -*count* bits. The result is always of the same sign as *n*. Shifting left (right) is a fast way to multiply (divide) a number by a power of 2.

The library function ASH is similar to SHIFT except that it operates as if negative integers are stored in "half-infinite" two's-complement notation.

```
(DEFUN SHIFT (N COUNT)
  ((AND (INTEGERP N) (INTEGERP COUNT))
    ((ZEROP COUNT) N)
    ((PLUSP COUNT)
      (SHIFT (* N 2) (SUB1 COUNT)) )
    (SHIFT (TRUNCATE N 2) (SUB1 COUNT)) )
  (BREAK (LIST 'SHIFT N COUNT)
         "Noninteger Argument") )

(DEFUN ASH (N COUNT)
  ((OR (>= N 0) (>= COUNT 0))
    (SHIFT N COUNT) )
  (ASH (SHIFT (SUB1 N) -1) (ADD1 COUNT)) )

$ (SHIFT 5 -1)                          ;halve 5
2

$ (SHIFT 5 1)                           ;double 5
10

$ (SHIFT 5 2)                           ;quadruple 5
20

$ (SHIFT -7 -2)                         ;logical right shift
-1

$ (ASH -7 -2)                           ;arithmetic right shift
-2
```

INTEGER-LENGTH [*n*] Function

If *n* is an integer, (INTEGER-LENGTH *n*) returns the number of bits required to represent the magnitude *n*.

```
(DEFUN INTEGER-LENGTH (N)
  ((INTEGERP N)
    ((ZEROP N) 0)
    (ADD1 (INTEGER-LENGTH (SHIFT N -1))) )
  (BREAK (LIST 'INTEGER-LENGTH N)
         "Noninteger Argument") )

$ (INTEGER-LENGTH 0)
0

$ (INTEGER-LENGTH 1)
1

$ (INTEGER-LENGTH 4)
3

$ (INTEGER-LENGTH 7)
3

$ (INTEGER-LENGTH 8)
4

$ (INTEGER-LENGTH -8)
4
```

12.8 Random Numbers

RANDOM [*n*, *state*] Library Function

If *n* is a positive integer, (RANDOM *n*) returns an approximately uniformly distributed random integer *x* such that $0 \leq x < n$. If *n* is a number but not a positive integer, (RANDOM *n*) returns an approximately uniformly distributed random rational *x* such that $0 \leq x < 1$, subject to the current precision as set by PRECISION.

The non-negative integer value of the random state variable *RANDOM-STATE* is used and maintained by RANDOM as the present state of the pseudo-random-number generator. If *state* is zero or a positive integer, (RANDOM *n state*) sets *RANDOM-STATE* to *state* before returning (RANDOM *n*).

```
(DEFUN RANDOM (N STATE)
  (IF (RANDOM-STATE-P STATE)
      (SETQ *RANDOM-STATE* STATE) )
  ((AND (INTEGERP N) (PLUSP N))
    (LET ((TEMP 0)
          (COUNT 0))
      (LOOP
        (SETQ *RANDOM-STATE* (LOGAND 4294967295
            (ADD1 (* 3141592653 *RANDOM-STATE*))))
        (SETQ TEMP (+ (* TEMP 32) *RANDOM-STATE*))
        ((> (INCQ COUNT 32) (+ (BITLENGTH N) 10))
          (SHIFT (* N TEMP) (- COUNT)) ) ) ) )
  ((NUMBERP N)
    (LET ((TEMP
            (EXPT 2 (- (* 16 (ADD1 (PRECISION)))
                       10))))
      (/ (* (ABS N) (RANDOM TEMP)) TEMP) ) ) )

$ (RANDOM 100)                    ;Random integer less than 100
84                                ;Result will vary

$ (RANDOM -100)                   ;Random number less than 100
61.2102272                        ;Result will vary
```

RANDOM-STATE — Library Control variable
MAKE-RANDOM-STATE [*state*] — Library Function

RANDOM-STATE is used to store the current state of muLISP's pseudo-random-number generator. The function RANDOM automatically updates the value of *RANDOM-STATE* when generating random numbers. *RANDOM-STATE* should only be set to values returned by the function MAKE-RANDOM-STATE. Initially *RANDOM-STATE* is set to (MAKE-RANDOM-STATE T).

If *state* is T, (MAKE-RANDOM-STATE *state*) returns a random-number state based on the system clock. If *state* is NIL or omitted, MAKE-RANDOM-STATE returns *RANDOM-STATE*. If *state* is an acceptable random-number state, MAKE-RANDOM-STATE returns *state*. Otherwise, MAKE-RANDOM-STATE returns NIL.

```
(DEFUN MAKE-RANDOM-STATE (STATE)
  ((NULL STATE)
    *RANDOM-STATE* )
  ((EQ STATE 'T)
    (TIME) )
  ((RANDOM-STATE-P STATE)
    STATE ) )

(SETQ *RANDOM-STATE* (MAKE-RANDOM-STATE 'T))

$ (SETQ *RANDOM-STATE* (MAKE-RANDOM-STATE 1000))
1000

$ (RANDOM 100)
45

$ (SETQ *RANDOM-STATE* (MAKE-RANDOM-STATE 1000))
1000

$ (RANDOM 100)                ;Resetting *RANDOM-STATE* makes
45                                    ;RANDOM results repeatable
```

RANDOM-STATE-P [*object*] Library Function

If *object* is an acceptable random-number state object, (RANDOM-STATE-P *object*) returns T, otherwise it returns NIL. The acceptable random-number states are integers greater than or equal to zero.

```
(DEFUN RANDOM-STATE-P (OBJ)
  (AND (INTEGERP OBJ) (>= OBJ 0)) )

$ (RANDOM-STATE-P 546)          ;Positives are valid random states
T

$ (RANDOM-STATE-P (TIME))       ;TIME gives valid random states
T

$ (RANDOM-STATE-P -73)     ;Negatives are not valid random states
NIL

$ (RANDOM-STATE-P 12.7)         ;Ratios are not valid random states
NIL
```

5.12.9 Numerical Error Trap Handlers

Rather than generating an error break directly when a "Nonnumeric Argument" or "Zero Divide" error occurs, the functions +, -, *, /, ABS, GCD, LCM, NUMERATOR, and DENOMINATOR call their respective error trap handlers to generate the error break. By appropriately redefining these error trap handlers, muLISP programs can modify the servicing of these errors as may be required by an application.

+TRAP [*object*$_1$, *object*$_2$]	Function
-TRAP [*object*$_1$, *object*$_2$]	Function
*TRAP [*object*$_1$, *object*$_2$]	Function
/TRAP [*object*$_1$, *object*$_2$]	Function

+TRAP, -TRAP, *TRAP, and /TRAP generate "Nonnumeric Argument" error breaks for the functions +, -, *, and / respectively. /TRAP generates a "Zero Divide" error break if its second argument is 0.

```
(DEFUN +TRAP LST
  (BREAK (CONS '+ LST) "Nonnumeric Argument") )

(DEFUN -TRAP LST
  (BREAK (CONS '- LST) "Nonnumeric Argument") )

(DEFUN *TRAP LST
  (BREAK (CONS '* LST) "Nonnumeric Argument") )

(DEFUN /TRAP LST
  ((ZEROP (CADR LST))
    (BREAK (CONS '/ LST) "Zero Divide") )
  (BREAK (CONS '/ LST) "Nonnumeric Argument") )
```

ABSTRAP [*object*] Function
NUMTRAP [*object*] Function
DENTRAP [*object*] Function
GCDTRAP [$object_1$, $object_2$] Function
LCMTRAP [$object_1$, $object_2$] Function

ABSTRAP, GCDTRAP, LCMTRAP, NUMTRAP, and DENTRAP generate "Nonnumeric Argument" error breaks for the functions ABS, GCD, LCM, NUMERATOR, and DENOMINATOR respectively.

```
(DEFUN ABSTRAP LST
  (BREAK (CONS 'ABS LST) "Nonnumeric Argument") )

(DEFUN NUMTRAP LST
  (BREAK (CONS 'NUMERATOR LST)
         "Nonnumeric Argument") )

(DEFUN DENTRAP LST
  (BREAK (CONS 'DENOMINATOR LST)
         "Nonnumeric Argument") )

(DEFUN GCDTRAP LST
  (BREAK (CONS 'GCD LST) "Nonnumeric Argument") )

(DEFUN LCMTRAP LST
  (BREAK (CONS 'LCM LST) "Nonnumeric Argument") )
```

5.13 INPUT PRIMITIVES

The input functions enable muLISP programs to read in and interpret data from the console and/or from open input files. Macro characters invoke special parse properties when a macro character is encountered by the function READ or READ-FROM-STRING. The input control variables control such things as the input radix base, the conversion of input to upper-case, expanding tabs, etc.

5.13.1 Input Functions

The input functions read and return data from an input source, either the console or an open input file. If the *source* argument to an input function is NIL or omitted, input is received from the Standard Input Source (SIS) (see Section 5.15). If the *source* argument is T, input is received from the console. If the *source* argument is an open input file, input is received from the file.

If a file is the input source, the following end-of-file error processing occurs when an input function attempts to read past the end of the file: If the *eof-value-p* argument to the function is NIL or omitted, an "End-Of-File" error break is generated (see Section 4.5). If the *eof-value-p* is nonNIL, the *eof-value* argument to the function is returned.

If the console is the input source, the input functions *other* than READ-BYTE invoke the muLISP line editor when all the characters have been read from the current line. Characters are echoed on the console screen as they are typed. Until RETURN is entered, the line editing commands described in Chapter 2 can be used to edit the line. This line edit mode of console input provides a uniform interface suitable for most standard applications.

If the console is the input source, the input function READ-BYTE returns the byte value of the keyboard character as soon as a key is pressed. The character is *not* echoed on the console screen. This raw mode of console input is useful for applications that need to process characters as soon as they are entered by the user.

READ [*source*, *eof-value-p*, *eof-value*] Function

READ reads one complete symbolic expression from the specified input source and returns the equivalent object. Well-formed expressions using either the list notation, dot notation, or a combination of both are acceptable input for READ.

Whitespace characters serve only to delimit tokens being read and are otherwise ignored by READ. The default whitespace characters are SPACE, RETURN, LINEFEED, PAGE and TAB. The ASCII code for RETURN is 0DH, LINEFEED is 0AH, PAGE is 0CH, and TAB is 09H.

Escape characters defeat the syntax properties of whitespace, macro, and escape characters so that they can be included as part of a token being read by READ. A single escape character delimits the character immediately following it. A multiple escape character delimits the characters between it and the next multiple escape character. A single escape character can be used to include single and multiple escape characters in the characters delimited by multiple escape characters. The default single escape character is the \ character. The default multiple escape character is the | character.

Macro characters invoke their macro definition function when the character is read by READ (see Section 5.13.2).

```
$ (READ)    (A (B . C) D)                    ;Read a list
(A (B . C) D)

$ (READ)    |This is delimited by multiple escapes|
|This is delimited by multiple escapes|

$ (READ)    '(+ 2 3)                         ;Read a macro character
(QUOTE (+ 2 3))
```

READ-ATOM [*source*, *eof-value-p*, *eof-value*] Function

READ-ATOM reads a token from the specified input source and returns the corresponding atom. A token is a string of characters delimited by either whitespace or break characters. READ-ATOM returns a number if the token begins with a decimal digit less than the current radix base. Otherwise, READ-ATOM returns a symbol. Number tokens are terminated by any character that is not a digit in the current radix base.

Whitespace characters serve only to delimit tokens and are not returned by READ-ATOM as atoms. READ-ATOM uses the same characters for whitespace characters as does READ. The default whitespace characters are SPACE, RETURN, LINEFEED, PAGE and TAB. The function COPY-CHAR-TYPE can be used to change the characters READ-ATOM recognizes as whitespace characters.

Break characters also delimit tokens, but unlike whitespace characters, they are returned by READ-ATOM as single character symbols. The default break characters are:

```
! " # $ % & ' ( ) * + , - . / :
< = > ? @ [ \ ] ^ _ ' { | } ~
```

READ-ATOM recognizes both single and multiple escape characters as a means of including whitespace and/or break characters as part of a token. See the discussion following READ for details on the use of escape characters.

```
$ (READ-ATOM) 123CAT                ;Read and return 123
123                                 ;CAT remains for input
```

SET-BREAK-CHARS [*list*, *flag*] Function
GET-BREAK-CHARS [] Function

If *flag* is NIL or omitted, (SET-BREAK-CHARS *list flag*) makes the characters in *list* the set of break characters accepted by READ-ATOM. If *flag* is nonNIL, SET-BREAK-CHARS adds the characters in *list* to the current set of break characters. SET-BREAK-CHARS always returns NIL.

(GET-BREAK-CHARS) returns a list of the current set of break characters.

The function UNBREAK-CHAR, defined in this example, makes a character a nonbreak character.

```
$ (DEFUN UNBREAK-CHAR (CHAR)
    (SET-BREAK-CHARS
        (DELETE CHAR (GET-BREAK-CHARS))) )

$ (UNBREAK-CHAR '-)                    ;Make - a nonbreak char
NIL

$ (READ-ATOM) TOP-LEVEL      ;Read TOP-LEVEL as one symbol
TOP-LEVEL
```

READ-LINE [*source*, *eof-value-p*, *eof-value*] Function
READ-CHAR [*source*, *eof-value-p*, *eof-value*] Function
READ-BYTE [*source*, *eof-value-p*, *eof-value*] Function

READ-LINE reads characters from the specified input source until a RETURN is encountered and returns the symbol whose print name consists of all the characters read except for the RETURN. If there is a LINEFEED immediately following the RETURN, it is discarded along with the RETURN. Note that READ-LINE does *not* discard blanks, recognize comments, or convert lower-case letters to upper-case.

READ-CHAR reads the next character from the specified input source and returns it as the symbol whose print name consists of that character. Note that READ-CHAR always returns a symbol, even if the character it reads is a digit.

READ-BYTE reads the next byte from the specified input source and returns it in the form on an integer between 0 and 255 inclusive. Note that "End-Of-File" error processing occurs if READ-BYTE attempts to read past the *physical* end of a file but *not* if it reads the *logical* end-of-file character (i.e. the byte 1AH).

```
$ (READ-LINE) One small step for man
| One small step for man|

$ (READ-CHAR)a                          ;Read the letter a
\a                                      ;Return the symbol a

$ (READ-BYTE)a                          ;Read the letter a
97                                      ;Return the ASCII code for a
```

UNREAD-CHAR [*source*] Function
PEEK-CHAR [*peek-type*, *source*, *eof-value-p*, *eof-value*] Function

UNREAD-CHAR restores the last character or byte read from the specified input source to the head of the input source, and returns NIL. Since only the most recently read character or byte can be restored, calling UNREAD-CHAR a second time on the same input source without an intervening read operation has no additional effect.

(PEEK-CHAR) reads the next character from the specified input source. (PEEK-CHAR T) reads characters from the specified input source until a non-whitespace character is encountered. If *character* is a symbol, (PEEK-CHAR *character*) reads characters from the specified input source until a character equal to the first character of *character*'s print name is encountered. In all the cases, PEEK-CHAR restores the last character read from the specified input source to the head of the input source, and returns the symbol whose print name consists of that character.

```
$ (PROG1 (READ-CHAR) (UNREAD-CHAR)) COMPUTER
C                              ;Read "C", unread "C", and return "C"

$ COMPUTER              ;Read and displayed by the read-eval-print loop

$ (PEEK-CHAR) COMPUTER
C                              ;Read "C", unread "C", and return "C"

$ COMPUTER              ;Read and displayed by the read-eval-print loop

$ (PEEK-CHAR T)          COMPUTER
C                      ;Read till non-whitespace character encountered

$ COMPUTER              ;Read and displayed by the read-eval-print loop

$ (PEEK-CHAR 'M) COMPUTER
M                                        ;Read till "M" encountered

$ MPUTER                ;Read and displayed by the read-eval-print loop
```

LISTEN [*source*] Function

If the specified input source has a character available for input, LISTEN returns T, otherwise it returns NIL. Specifically, if the console is the input source, LISTEN returns T if a character is available for input from the keyboard (i.e. a key has been pressed). If a file is the input source, LISTEN returns T if a character is available for input from the file (i.e. the read position is *not* at the end of the file).

LISTEN is used in this example to determine if the user has entered a character from the keyboard. If not, a beep character (i.e. ASCII 7) is periodically sent to the console to let the user know that the program is waiting (impatiently) for input. When the user does press a key, BEEP-READ-BYTE uses READ-BYTE to read and return the keyboard byte.

```
$ (DEFUN BEEP-READ-BYTE (
    _ CTR )                          ;Make CTR a local variable
    (CLEAR-INPUT T)                  ;Empty console input buffer
    (SETQ CTR 0)                     ;Initialize CTR
    (LOOP
      ((LISTEN T)                    ;Key pressed?
        (READ-BYTE T) )              ;Read key
      (TONE NIL 100)                 ;Pause 1/10 second
      (INCQ CTR)                     ;Increment CTR
      (IF (= CTR 50)                 ;5 seconds elapsed?
          (PROGN
            (SETQ CTR 0)             ;Reinitialize CTR
            (WRITE-BYTE 7 NIL T))) ) )   ;Beep console
```

CLEAR-INPUT [*source*] Function

If the console is the specified input source, CLEAR-INPUT clears the console keyboard buffer and returns NIL. Otherwise, CLEAR-INPUT just returns NIL.

After a program detects an error and before querying the user on how to proceed, CLEAR-INPUT is useful for emptying the keyboard buffer of now extraneous input. The following function clears the console input buffer before reading a byte from the keyboard. See the formal definition of Y-OR-N-P for another example using CLEAR-INPUT.

```
$ (DEFUN CLEAR-READ-BYTE ()
    (CLEAR-INPUT T)
    (READ-BYTE T) )
```

READ-FROM-STRING [*string*, *eof-value-p*, *eof-value*] Function
STRING-INDEX Variable
READ-STRINGS Variable

If *strings* is a nonempty list of symbols, (READ-FROM-STRING *strings*) reads one complete symbolic expression from the symbols in *strings*, sets *READ-STRINGS* to the tail of *strings* beginning with the last symbol read, sets *STRING-INDEX* to 1 plus the index in the last symbol read of the last character read, and returns the object equivalent to the expression read. A RETURN character is inserted in the input stream between each string read by READ-FROM-STRING.

If *string* is a symbol, (READ-FROM-STRING *string*) is equivalent to (READ-FROM-STRING (LIST *string*)).

READ-FROM-STRING accepts well-formed expressions expressed using either list notation, dot notation, or a combination of both. READ-FROM-STRING processes whitespace, escape, and macro characters the same way READ processes them. Also, READ-FROM-STRING processes attempts to read past the end of a string the same way READ processes attempts to read past the end of a file.

```
$ (READ-FROM-STRING "(1 2 3)")
(1 2 3)

$ *STRING-INDEX*
7

$ (READ-FROM-STRING "apple orange")
APPLE

$ *STRING-INDEX*
5

$ (READ-FROM-STRING '("(1 2"  "3 4)"  "5 6"))
(1 2 3 4)

$ *READ-STRINGS*
(|3 4)| |5 6|)
```

PARSE-INTEGER [*string*, *radix*] Library Function
STRING-INDEX Variable

If *radix* is an integer between 2 and 36 inclusive, (PARSE-INTEGER *string radix*) skips any leading whitespace, attempts to read and parse an integer in base *radix* from *string*, sets *STRING-INDEX* to 1 plus the index of the last valid digit character it read, and returns the value of the integer it parsed. *radix* defaults to ten.

PARSE-INTEGER accepts integers consisting of an optional plus (+) or minus (-) sign followed by one or more digits in base *radix*. It stops reading and returns the value of the integer read when an invalid digit or the end of *string* is encountered. If the first non-whitespace characters of *string* do not constitute a valid integer, PARSE-INTEGER returns NIL.

```
$ (PARSE-INTEGER "123")
123

$ *STRING-INDEX*
3

$ (PARSE-INTEGER "  -0054321+123")
-54321

$ *STRING-INDEX*
10
```

5.13.2 Macro Characters

Macro characters invoke their macro definition function when they are encountered by the functions READ and READ-FROM-STRING. Not only do macro characters make it possible to abbreviate frequently used code segments, but they can alter the syntax used in LISP source files to better suit the application at hand. Macro characters can be categorized as being either terminating, non-terminating, or comment.

Terminating and non-terminating macro characters insert exactly one object into the input stream when they are encountered. Comment macro characters insert no object into the input stream. Comment and terminating macro characters will terminate a token being read by READ. Non-terminating macro characters do not terminate a token being read and hence must be preceded by at least one whitespace character in order to be recognized as a macro character.

muLISP primitively defines several of the most commonly used macro characters. The function COPY-CHAR-TYPE can be used to assign the definition of these macro characters to other characters. The function SET-MACRO-CHAR can be used to define new macro characters.

COPY-CHAR-TYPE [$char_1$, $char_2$, *flag*] Function

If *flag* is nonNIL, (COPY-CHAR-TYPE $char_1$ $char_2$ *flag*) copies the character type of $char_2$ in the current readtable to $char_1$. If *flag* is NIL or omitted, COPY-CHAR-TYPE copies the character type of $char_2$ in the default readtable to $char_1$. If $char_2$ is a macro character, a pointer to its macro definition is also copied to $char_1$. If successful, COPY-CHAR-TYPE returns T, otherwise it returns NIL.

```
$ (COPY-CHAR-TYPE '\~ '\;)          ;Make ~ a comment char
T
```

SET-MACRO-CHAR [*char*, *definition*, *flag*] Function
GET-MACRO-CHAR [*char*, *flag*] Function

If *flag* is NIL or omitted, (SET-MACRO-CHAR *char definition flag*) uses *definition* to establish *char* as a terminating macro character. If *flag* is the symbol COMMENT, SET-MACRO-CHAR establishes *char* as a comment macro character. Otherwise, SET-MACRO-CHAR establishes *char* as a non-terminating macro character.

If *flag* is NIL or omitted, (GET-MACRO-CHAR *char flag*) returns the macro character definition associated with *char*. If *flag* is nonNIL, GET-MACRO-CHAR returns LAMBDA if *char* is a macro character. In either case, it returns NIL if *char* is not a macro character.

The following example shows how to define the percent sign (%) as a comment macro that recognizes text enclosed in matching percent signs as a comment. Note that unlike comments delimited by a semicolon, comments delimited by percent signs can be more than a single line and can have noncommented expressions to their right as well as left:

```
$ (SET-MACRO-CHAR '\%                    ;Define % comment macro
    '(LAMBDA ()
      (PEEK-CHAR '\%)
      (READ-CHAR) )
    'COMMENT)

$ '(DOG CAT % This is a comment. % COW)
(DOG CAT COW)
```

((left parenthesis) Terminating macro character

The left parenthesis macro reads expressions until a matching right parenthesis, right square bracket, or dot is encountered. If a right parenthesis is encountered, the macro returns a list of the expressions read. If a right square bracket is encountered, the macro "unreads" the bracket so that it can be read again and the macro returns a list of the expressions read. This makes the right square bracket a super-parenthesis that closes all left parentheses back to the top-level READ.

If a dot is encountered, the macro reads one more expression and checks to ensure that the next non-whitespace character read is either a right parenthesis or a right square bracket. If so, the macro returns a list of the expressions read before the dot was encountered, nconced with the expression read after the dot. If not, a "Syntax Error" break occurs.

) (right parenthesis) Terminating macro character
] (right square bracket) Terminating macro character

The left parenthesis macro recognizes the right parenthesis and the right square bracket as list terminators. The right parenthesis macro and the right square bracket macro generate a "Syntax Error" break if too many right parentheses or right square brackets are encountered by READ.

, (comma) Terminating macro character

The comma macro generates a "Syntax Error" break if a comma is encountered directly by READ.

' (single quote) Terminating macro character

The single quote mark macro reads in the expression following the mark and returns a two element list consisting of the symbol QUOTE and the expression just read. The single quote simplifies the entry of constants in LISP expressions.

; (semicolon) Comment macro character

The semicolon macro reads in characters until the end of line character RETURN is encountered. Since no object is inserted into the input stream, this effectively makes semicolon a single-line comment delimiter.

" (double quote) Terminating macro character

The double quote macro reads characters from the input stream until a matching double quote character is encountered and returns the symbol whose print name consists of the characters read. Single escape characters can be used to include double quote characters and single escape characters in the string.

` (backquote) Library terminating macro character

When defining macros using DEFMACRO (see Section 5.10), executable code must be created for evaluation at runtime. This code can be created directly using constructor functions such as CONS, APPEND, LIST, and LIST*. However if the code to be generated is at all complex, macros defined in this way are difficult to understand and debug.

The backquote macro character facility can greatly simplify macro definitions. It lets you create code or other complex data structures by giving a "template" for the desired data structure. Also you can specify variables in the template so that they will be filled in by their values.

An expression preceded by a comma is a variable whose value must be inserted in the template at that position. An expression preceded by a comma and an at-sign (@) is a variable whose value must be *spliced* into the list in the template in which the expression occurs. Section 22.1.3 of the book Common LISP, The Language (Steele [1984]) contains a more detailed explanation of the backquote macro.

The following example illustrates the code created by the backquote macro and the constructor functions necessary to produce the equivalent code directly:

```
$ (SETQ X 7 Y '(2 3 5))
(2 3 5)

$ '(IF (NUMBERP ,X) (+ ,X ,@Y))
(IF (NUMBERP 7) (+ 7 2 3 5))

$ (LIST 'IF (LIST 'NUMBERP X) (LIST* '+ X Y))
(IF (NUMBERP 7) (+ 7 2 3 5))
```

5.13.3 Input Control Variables

The input control variables extend the flexibility of the input functions by making several input options available to the user. Most operate as switches that are either OFF (i.e. NIL) or ON (i.e. nonNIL).

INPUT-ECHO: NIL — Control variable

If *INPUT-ECHO* is nonNIL, the characters read from files are echoed to the standard output sink (SOS). Note that since comments are also echoed, English language text within a comment can be displayed conveniently without muLISP having to actually process the text.

READ-BASE: 10 — Control variable

If *READ-BASE* is an integer between 2 and 36 inclusive, that integer is the radix base assumed by READ, READ-FROM-STRING, and READ-ATOM when numbers are read in. Otherwise, these functions assume numbers are expressed in decimal (i.e. radix base ten).

READ-UPCASE: T — Control variable

If *READ-UPCASE* is nonNIL, lower-case letters read in by READ, READ-FROM-STRING, and READ-ATOM are converted to upper-case unless the letters are part of a string, or are delimited by single or multiple escape characters. If *READ-UPCASE* is NIL, lower-case letters are not converted as they are read in. Note that regardless of the value of *READ-UPCASE*, print names already in the system containing lower-case letters are not changed.

TAB-EXPAND: NIL Control variable

If *TAB-EXPAND* is nonNIL, a TAB character is expanded to between one and eight spaces when it is read by an input function other than READ-BYTE. If *TAB-EXPAND* is NIL, the TAB character is not expanded. Regardless of the value of *TAB-EXPAND*, READ-BYTE returns 9 when the TAB character is read. The variable *BLANK-COMPRESS* controls the converse operation of compressing of BLANK characters to a single TAB character (see Section 5.14).

IGNORE-CASE: NIL Control variable

When a print name read by either the function READ, READ-FROM-STRING, or READ-ATOM is being interned, muLISP first attempts to find and return a symbol in the current oblist that has the same print name.

If no such symbol can be found, and *IGNORE-CASE* is NIL *or* a file is the input source, a symbol having the new print name is added to the oblist. If *IGNORE-CASE* is nonNIL *and* the console is the input source, muLISP ignores the case of letters while attempting to find a symbol in the oblist that has the same print name as the print name being interned. If a matching symbol is still not found, a symbol having the new print name is added to the oblist. Note that *IGNORE-CASE* only affects the functions READ, READ-FROM-STRING, and READ-ATOM.

5.14 OUTPUT PRIMITIVES

The output functions enable muLISP programs to format and send data to the console and/or to open output files. The output control variables control such things as the output radix base, the conversion of output to lower-case, compressing blanks to tabs, the use of escape characters, etc.

5.14.1 Output Functions

The output functions format and send data to an open output sink, either the console or an open output file. If the *sink* argument to an output function is NIL or omitted, output is sent to the Standard Output Sink (SOS) (see Section 5.15). If the *sink* argument is T, output is sent to the console. If the *sink* argument is an open output file, output is sent to the file.

If a file is the output sink, the following disk full error processing occurs when an output function attempts to write data to a disk file and there is insufficient space on the disk to store the data: A throw of NIL is made to the label "Disk Full". If this throw is not caught by an application program, the muLISP top-level loop (see Section 5.19) catches the throw and generates a "Disk Full" error break (see Section 4.8).

WRITE [*object*, *sink*] Function

(WRITE *object sink*) sends the printed representation of *object* to *sink* and returns *object*. WRITE uses the value of the output control variables (see Section 5.14.2) to determine how atoms in the object are represented.

Symbols are represented by the string of characters in their print name. The variable *PRINT-ESCAPE* controls the use of escape characters around print names containing whitespace, macro, or escape characters. The variable *PRINT-DOWNCASE* controls the conversion of upper-case letters to lower-case.

Numbers are represented by the string of numerals equivalent to the number. The variable *PRINT-BASE* controls the radix base used to represent numbers. The variable *PRINT-POINT* controls the maximum number of digits used to display ratios.

Conses are represented using list notation where possible and dot notation where necessary (see Chapter 4).

```
(DEFUN WRITE (OBJ SINK)
  (send the printed representation of OBJ to SINK)
  OBJ )
```

PRIN1 [*object*, *sink*] Function
PRINT [*object*, *sink*] Function
PRINC [*object*, *sink*] Function

Using escape characters as necessary, (PRIN1 *object sink*) sends the printed representation of *object* to *sink* and returns *object*. PRIN1 is equivalent to WRITE with *PRINT-ESCAPE* temporarily bound to T. In short, PRIN1 produces output suitable for input by READ.

Using escape characters as necessary, (PRINT *object sink*) sends the printed representation of *object* to *sink*, then it sends the ASCII newline character pair (see TERPRI) to *sink*, and finally it returns *object*. PRINT is equivalent to calling PRIN1 and then TERPRI.

Using *no* escape characters, (PRINC *object sink*) sends the printed representation of *object* to *sink* and returns *object*. PRINC is equivalent to WRITE with *PRINT-ESCAPE* temporarily bound to NIL. In short, PRINC produces output more suitable for human readers.

```
(DEFUN PRIN1 (OBJ SINK
    *PRINT-ESCAPE* )
  (SETQ *PRINT-ESCAPE* T)
  (WRITE OBJ SINK) )

(DEFUN PRINT (OBJ SINK)
  (PRIN1 OBJ SINK)
  (TERPRI 1 SINK)  OBJ )

(DEFUN PRINC (OBJ SINK
    *PRINT-ESCAPE* )
  (SETQ *PRINT-ESCAPE* NIL)
  (WRITE OBJ SINK) )

$ (PRINT '(A B C))                    ;Print and return list
(A B C)
(A B C)

$ (PRINC |The quick brown fox|)    ;Print and return phrase
The quick brown fox|The quick brown fox|
```

WRITE-STRING [*symbol*, *sink*] Function
WRITE-LINE [*symbol*, *sink*] Function

If *symbol* is a symbol object, (WRITE-STRING *symbol sink*) sends the characters in *symbol*'s print name to *sink* and returns *symbol*. If *symbol* is not a symbol object, WRITE-STRING returns NIL.

If *symbol* is a symbol object, (WRITE-LINE *symbol sink*) sends the characters in *symbol*'s print name to *sink*, then it sends the ASCII newline character pair (see TERPRI) to *sink*, and finally it returns *symbol*. If *symbol* is not a symbol object, WRITE-LINE returns NIL.

Both WRITE-STRING and WRITE-LINE ignore the value of the control variables *PRINT-ESCAPE* and *PRINT-DOWNCASE* and output the characters exactly as they occur in *symbol*'s print name.

```
(DEFUN WRITE-STRING (SYM SINK)
  ((SYMBOLP SYM)
    (send OBJ'S printed representation to SINK)
    SYM ) )

(DEFUN WRITE-LINE (SYM SINK)
  ((SYMBOLP SYM)
    (WRITE-STRING SYM SINK)
    (TERPRI)
    SYM ) )

$ (WRITE-STRING |muLISP|)
muLISP|muLISP|                    ;Write and return "muLISP"

$ (WRITE-LINE |muLISP|)
muLISP                            ;Write "muLISP" and a newline
|muLISP|                                ;Return "muLISP"
```

TERPRI [*n*, *sink*] Function
FRESH-LINE [*sink*] Function

If *n* is zero or a positive integer, (TERPRI *n sink*) sends *n* ASCII newline character pairs (i.e. RETURN and LINEFEED) to *sink* and returns NIL. *n* defaults to 1.

If a newline character pair was *not* the last thing sent to *sink*, (FRESH-LINE *sink*) sends a newline to *sink* and returns T. Otherwise, FRESH-LINE simply returns NIL.

```
(DEFUN TERPRI (N SINK)
  ((AND (INTEGERP N) (>= N 0))
    (LOOP
      ((ZEROP N) NIL)
      (WRITE-BYTE 13 1 SINK)
      (WRITE-BYTE 10 1 SINK)
      (DECQ N) ) )
  (TERPRI 1 SINK) )

(DEFUN FRESH-LINE (SINK)
  ((ZEROP (SPACES SINK)) NIL)
  (TERPRI SINK)
  T )

$ (PROGN (PRIN1 'MARS) (TERPRI 2) 'VENUS)
MARS                                    ;Print MARS
                                        ;Move down 2 lines
VENUS                                   ;Print VENUS
```

SPACES [*n*, *sink*] Function

If *n* is zero or a positive integer, (SPACES *n sink*) sends *n* ASCII SPACE characters to *sink* and returns the number of characters sent since the last newline character pair was sent. *n* defaults to 0.

```
(DEFUN SPACES (N SINK)
  ((AND (INTEGERP N) (>= N 0))
    (LOOP
      ((ZEROP N)
        (characters sent since last newline) )
      (WRITE-BYTE 32 1 SINK)
      (DECQ N) ) )
  (SPACES 0 SINK) )

$ (PROGN (PRIN1 'MARS) (SPACES 8) 'VENUS)
MARS        VENUS          ;Print VENUS 8 spaces after MARS

$ (DEFUN TAB (N)                    ;Define a tab function
    (SPACES (- N (SPACES))) )
TAB

$ (PROGN (PRIN1 'MARS) (TAB 8) 'VENUS)
MARS    VENUS                       ;Print VENUS at column 8
```

WRITE-BYTE [*n*, *m*, *sink*] Function

If *n* is an integer between 0 and 255 inclusive and *m* is zero or a positive integer, (WRITE-BYTE *n m sink*) writes *m* bytes of *n* to *sink* and returns *n*. *m* defaults to 1. If *n* is not within range, WRITE-BYTE returns NIL.

```
(DEFUN WRITE-BYTE (N M SINK)
  ((AND (INTEGER N) (<= 0 N 255))
    ((AND (INTEGER M) (>= M 0))
      (LOOP
        ((ZEROP M) N)
        (write N to SINK)
        (DECQ M) ) )
    (WRITE-BYTE N 1 SINK) ) )

$ (WRITE-BYTE 65)
A65                                   ;Write "A" and return 65

$ (WRITE-BYTE 65 10)
AAAAAAAAAA65                          ;Write 10 "A"s and return 65

$ (WRITE-BYTE 10)
                                      ;Write a LINEFEED
10                                          ;Return 10
```

5.14.2 Output Control Variables

The output control variables enhance the control muLISP programs have over character output. Most operate as switches that are either OFF (i.e. NIL) or ON (i.e. nonNIL).

AUTO-NEWLINE: T — Control variable

If *AUTO-NEWLINE* is nonNIL and a file is the output sink, newline characters are automatically sent to the file when necessary to enforce the line length of the file. If *AUTO-NEWLINE* is nonNIL and the console is the output sink, newline characters are automatically sent to the console after a character is displayed in the right-most column of the current window (see Section 5.21).

If *AUTO-NEWLINE* is NIL and a file is the output sink, the line length of the file is *not* enforced. If *AUTO-NEWLINE* is nonNIL and the console is the output sink, the cursor is left at the right-most column of the window, and new characters simply overlay previous ones.

BLANK-COMPRESS: NIL — Control variable

If *BLANK-COMPRESS* is nonNIL, when two to eight BLANK characters ending on a tab stop are encountered by an output function other than WRITE-BYTE, they are compressed to a single TAB character before being sent to the output sink. If *BLANK-COMPRESS* is NIL, the BLANK characters are not compressed. Regardless of the value of *BLANK-COMPRESS*, (WRITE-BYTE 32) sends a BLANK character to the output sink. The variable *TAB-EXPAND* controls the converse operation of expanding a TAB character to multiple SPACE characters (see Section 5.13).

OUTPUT-ECHO: NIL — Control variable

If *OUTPUT-ECHO* is nonNIL and a disk file is the output sink, characters sent to the file are also sent to the console. If *OUTPUT-ECHO* is NIL, characters are not echoed on the console.

PRINTER-ECHO: NIL Control variable

If *PRINTER-ECHO* is nonNIL, characters sent to the console are also sent to the system's printer. This is a convenient means of producing a "hard copy" record of a muLISP session. IF *PRINTER-ECHO* is NIL, no printer echoing occurs.

PRINT-DOWNCASE: NIL Control variable

If *PRINT-DOWNCASE* is nonNIL, upper-case letters are converted to lower-case as they are sent to an output sink by the functions WRITE, PRIN1, PRINT, and PRINC. Note that this case conversion is only done as the characters are being output and does not affect the actual print name stored in memory. If *PRINT-DOWNCASE* is NIL, no case conversion occurs.

```
$ (SETQ *PRINT-DOWNCASE* T)
T

$ !muLISP
mulisp

$ (SETQ *PRINT-DOWNCASE* NIL)
NIL

$ 'muLISP
MULISP
```

PRINT-ESCAPE: T Control variable

If *PRINT-ESCAPE* is nonNIL and a symbol has a print name that contains lower-case letters, whitespace or macro or escape characters (see Section 5.13), or begins with a decimal digit, the symbol's print name is delimited by escape characters when the characters in the symbol's print name are sent to an output sink by the function WRITE. If *PRINT-ESCAPE* is NIL, the print name of such symbols is not delimited by escape characters.

Delimiting the print name of such symbols by escape characters is essential for programs that generate muLISP source files, so the symbols can be read back in correctly by the functions READ and READ-ATOM.

```
$ (SETQ *PRINT-ESCAPE* NIL)          ;Turn off escape chars
NIL

$ '|I think, therefore I am.|        ;Display w/o escape chars
I think, therefore I am.

$ (SETQ *PRINT-ESCAPE* T)            ;Turn on escape chars
T

$ '|I think, therefore I am.|        ;Display with escape chars
|I think, therefore I am.|
```

PRINT-BASE: 10 Control variable

If *PRINT-BASE* is an integer between 2 and 36 inclusive, it is the radix base used to represent numbers sent to an output sink. Otherwise, numbers sent to an output sink are expressed in decimal (i.e. radix base ten).

```
$ (SETQ TEN 10)                          ;Initialize TEN to 10
10

$ (SETQ *PRINT-BASE* 16)          ;Print numbers in hexadecimal
10

$ TEN                                    ;Ten in hexadecimal
0A

$ (SETQ *PRINT-BASE* 2)             ;Print numbers in binary
10

$ TEN                                        ;Ten in binary
1010

$ (SETQ *PRINT-BASE* 10)            ;Print numbers in decimal
10
```

***PRINT-POINT*: 7** Control variable

If *PRINT-POINT* is a nonnegative integer, ratios are sent to an output sink using decimal notation with up to *PRINT-POINT* digits following the decimal point. Otherwise, ratios are sent using slash notation (i.e. the numerator, a "/", and the denominator).

Note that *PRINT-POINT* only affects the representation of numbers being sent to an output sink; it does not affect the actual precision of the number. Numbers can be entered using either decimal or slash notation regardless of the value of *PRINT-POINT*.

The initial value of *PRINT-POINT* is 7. Its initial value can be changed by modifying the muLISP base page (see Chapter 9 and Appendix B for details).

```
$ (SETQ PI 3.1416)                    ;Initialize PI to 3.1416
3.1416

$ (SETQ *PRINT-POINT* NIL)            ;Print using slash notation
NIL

$ PI                                  ;PI in slash notation
3927/1250

$ (SETQ *PRINT-POINT* 2)              ;Print up to 2 digits
2

$ PI                                  ;PI to 2 digits
3.14

$ (SETQ *PRINT-POINT* 7)              ;Print up to 7 digits
7

$ PI                                  ;PI to 7 digits
3.1416
```

5.15 FILE SYSTEM PRIMITIVES

The muLISP file system enables programs to open and close files for input and/or output, select a file for standard input and/or output, adjust the read and/or write position of open files, and perform standard file and directory operations. Files can be specified using their full MS-DOS pathnames. The only limit on the number of files that can be open at any one time is the limit imposed by MS-DOS (see the MS-DOS Reference Manual for details).

Most of the file system functions are given one or more file names as arguments and/or return one or more file names as values. A file name not only specifies a file's name and extension, but the drive and directory the file is on. muLISP uses the standard MS-DOS convention for naming files. For example,

```
C:\MULISP\ANIMAL.LSP
```

specifies the file ANIMAL.LSP on drive C in the directory MULISP. Since the backslash character (\) used in file names is also the muLISP multiple escape character (see Section 5.13), two backslashes must be entered for each backslash in the file's name. Thus, the above file name is entered as

```
C:\\MULISP\\ANIMAL.LSP
```

If no drive is specified in a file name given to a file system function, the MS-DOS default drive is assumed. If no directory is specified in a file name, the MS-DOS current directory is assumed. If no extension is specified in a file name, no extension name is assumed. Note that this is a change from earlier versions of muLISP, where a default extension name of LSP was assumed.

Certain file names are reserved by MS-DOS as system device names. These include CON, AUX, COM1, COM2, LPT1, LPT2, LPT3, PRN, and NUL (see the MS-DOS Manual for details). If a file system function is given one of these names as a file name, the corresponding system device will be used instead of a file.

5.15.1 Input File Primitives

The input file primitives enable programs to open and close files for input, select the Standard Input File (SIF) and the Standard Input Source (SIS), and adjust the read position of open input files.

If a call to an input function does not specify an input source (see Section 5.13), the SIS is used. The SIS can be either the console or the SIF. If there is a SIF, the value of *INPUT-FILE* determines if the SIS is the console or the SIF. Note that although several files can be open for input, only one input file at a time can be the SIF. If there is no SIF, the console is the SIS regardless of the value of *INPUT-FILE*.

OPEN-INPUT-FILE [*filename*] Function

If the file on the drive and directory specified by *filename* exists, (OPEN-INPUT-FILE *filename*) opens *filename* for input and makes it the SIF, initializes its read position to 0, sets the control variable *INPUT-FILE* to its pathname (thus making *filename* the SIS as well as the SIF), and returns its pathname. If no file specified by *filename* exists, OPEN-INPUT-FILE returns NIL.

The example below opens the file ANIMAL.LSP for input and makes it the SIF. Note that making ANIMAL.LSP the SIF does *not* automatically load the file. Instead use the function LOAD to read in and evaluate the expressions in a source file.

```
(DEFUN OPEN-INPUT-FILE (FILE-NAME)
  ((PROBE-FILE FILE-NAME)
    (open FILE-NAME for input)
    (INPUT-FILE FILE-NAME)
    (FILE-READ-POSITION 0)
    (SETQ *INPUT-FILE* (INPUT-FILE)) ) )

$ (OPEN-INPUT-FILE 'ANIMAL.LSP)        ;Open ANIMAL.LSP
C:ANIMAL.LSP

$ *INPUT-FILE*                         ;*INPUT-FILE* is now nonNIL
C:ANIMAL.LSP

$ (FILE-READ-POSITION)                 ;Initial read position
0

$ (READ-LINE)                          ;Read first line of file
|;File: ANIMAL.LSP                    Soft Warehouse, Inc.|

$ (FILE-READ-POSITION)                 ;Current read position
56
```

CLOSE-INPUT-FILE [*filename*] Function

If *filename* is an open input file, (CLOSE-INPUT-FILE *filename*) closes *filename* for input and returns T. If *filename* is also the SIF, CLOSE-INPUT-FILE sets the control variable *INPUT-FILE* to NIL. If *filename* is not an open input file, (CLOSE-INPUT-FILE *filename*) returns NIL.

If there is a SIF, (CLOSE-INPUT-FILE) closes the SIF for input; sets the control variable *INPUT-FILE* to NIL, and returns T. If there is no SIF, (CLOSE-INPUT-FILE) returns NIL.

```
(DEFUN CLOSE-INPUT-FILE (FILE-NAME)
  ((NULL FILE-NAME)
    ((INPUT-FILE)
      (CLOSE-INPUT-FILE (INPUT-FILE)) ) )
  ((INPUT-FILE-P FILE-NAME)
    (IF (EQ FILE-NAME (INPUT-FILE))
        (SETQ *INPUT-FILE*) )
    (close FILE-NAME for input)
    T ) )

$ (CLOSE-INPUT-FILE)                          ;Close the SIF
T

$ *INPUT-FILE*                          ;*INPUT-FILE* is now NIL
NIL
```

INPUT-FILE [*filename*] Function

If *filename* is an open input file, (INPUT-FILE *filename*) makes *filename* the SIF and returns its pathname. If *filename* is not an open input file, (INPUT-FILE *filename*) returns NIL.

If there is a SIF, (INPUT-FILE) returns the pathname of the SIF. If there is no SIF, (INPUT-FILE) returns NIL.

```
(DEFUN INPUT-FILE (FILE-NAME)
  ((NULL FILE-NAME)
    ((there is a SIF)
      (the pathname of the SIF) ) )
  ((INPUT-FILE-P FILE-NAME)
    (make FILE-NAME the SIF)
    (the pathname of FILE-NAME) ) )

$ (OPEN-INPUT-FILE 'ANIMAL.LSP)        ;Open ANIMAL.LSP
C:ANIMAL.LSP

$ (OPEN-INPUT-FILE 'HANOI.LSP)         ;Open HANOI.LSP
C:HANOI.LSP

$ (INPUT-FILE)                         ;HANOI.LSP is the SIF
C:HANOI.LSP

$ (INPUT-FILE 'ANIMAL.LSP)             ;Make ANIMAL.LSP the SIF
C:ANIMAL.LSP

$ (INPUT-FILE)                         ;ANIMAL.LSP is the SIF
C:ANIMAL.LSP
```

INPUT-FILE-P [*filename*] Function
INPUT-FILES [] Function

If *filename* is an open input file, (INPUT-FILE-P *filename*) returns T, otherwise it returns NIL.

(INPUT-FILES) returns a list of the pathnames of the currently open input files.

```
(DEFUN INPUT-FILE-P (FILE-NAME)
  ((if FILE-NAME is an open input file) T)
  NIL )

(DEFUN INPUT-FILES ()
  (a list of the open input files) )

$ (OPEN-INPUT-FILE 'ANIMAL.LSP)        ;Open ANIMAL.LSP
C:ANIMAL.LSP

$ (OPEN-INPUT-FILE 'HANOI.LSP)         ;Open HANOI.LSP
C:HANOI.LSP

$ (INPUT-FILE-P 'ANIMAL.LSP)              ;Open for input?
T                                                 ;Yes

$ (INPUT-FILE-P 'HANOI.LSP)               ;Open for input?
T                                                 ;Yes

$ (INPUT-FILES)                    ;Currently open input files
(C:HANOI.LSP C:ANIMAL.LSP)
```

INPUT-FILE: NIL Control variable

If there is a SIF and *INPUT-FILE* is nonNIL, standard input comes from the SIF; otherwise standard input comes from the console.

Normally the functions OPEN-INPUT-FILE and INPUT-FILE provide sufficient control over the source of character input to muLISP programs. However, after reading part of a file, it may be necessary to temporarily receive input from the console before returning to the file. The control variable *INPUT-FILE* provides this capability.

After using OPEN-INPUT-FILE to open a file and make it the SIF, the console can be made the SIS without changing the SIF by setting *INPUT-FILE* to NIL. A subsequent nonNIL assignment to *INPUT-FILE* will again make the SIF the SIS, and reading of the file will resume from the point at which it had been suspended.

An alternative method of temporarily receiving input from the console is to specify the console in calls on the input functions (see Section 5.13).

FILE-READ-POSITION [*n*, *filename*] Function

If *filename* is an open input file and *n* is zero or a positive integer, (FILE-READ-POSITION *n filename*) sets the read position of *filename* to *n* and returns T. If *n* is neither zero nor a positive integer, FILE-READ-POSITION returns the current read position of *filename*. If *filename* is not an open input file, FILE-READ-POSITION returns NIL. If no *filename* argument is given, the SIF is assumed.

A file's read position determines the point in the file at which the next character will be read. When a file is opened for input, the file's read position is set to 0 (the first byte of the file). After a character is read from the file, the file's read position is automatically incremented. Thus FILE-READ-POSITION needs to be used only when a file must be accessed randomly.

```
(DEFUN FILE-READ-POSITION (N FILE-NAME)
  ((NULL FILE-NAME)
    ((INPUT-FILE)
      (FILE-READ-POSITION N (INPUT-FILE)) ) )
  ((INPUT-FILE-P FILE-NAME)
    ((AND (INTEGERP N) (>= N 0))
      (set the read position of FILE-NAME to N)
      T )
    (FILE-NAME current read position) ) )
```

A good example of the use of FILE-READ-POSITION is in the muLISP Tutorial System source file LESSONS.LSP. The system reads and displays the script of a lesson using the function READ-LESSON. Since the system provides an option for viewing the previous screen, READ-LESSON pushes the read position marking the beginning of each screen onto the "stack" *POSN-LIST*. When the student wants to see the previous screen, the call

```
(FILE-READ-POSITION (CAR *POSN-LIST*))
```

is used to move the read position back to the beginning of the screen so it can be read and displayed again.

5.15.2 Output File Primitives

The output file primitives enable programs to open and close files for output, select the Standard Output File (SOF) and the Standard Output Sink (SOS), and adjust the write position and line length of open output files.

If a call to an output function does not specify an output sink (see Section 5.14), the SOS is used. The SOS can be either the console or the SOF. If there is a SOF, the value of *OUTPUT-FILE* determines if the SOS is the console or the SOF. Note that although several files can be open for output, only one output file at a time can be the SOF. If there is no SOF, the console is the SOS regardless of the value of *OUTPUT-FILE*.

OPEN-OUTPUT-FILE [*filename*, *overwrite*] Function

In the event *filename* is already an open output file, (OPEN-OUTPUT-FILE *filename overwrite*) first closes *filename* for output before proceeding with the following operations:

If the file on the drive and directory specified by *filename* exists and *overwrite* is nonNIL, (OPEN-OUTPUT-FILE *filename overwrite*) opens *filename* for output and makes it the SOF, initializes its write position to 0, sets the control variable *OUTPUT-FILE* to its pathname (thus making *filename* the SOS as well as the SOF), and returns its pathname.

If the file specified by *filename* does not exist and *overwrite* is nonNIL, (OPEN-OUTPUT-FILE *filename overwrite*) first creates a new directory entry for *filename* before proceeding with the above. The overwrite flag makes it possible to simply reopen and modify existing files rather than having to write an entirely new file.

If *overwrite* is NIL or not given, (OPEN-OUTPUT-FILE *filename overwrite*) opens a temporary file for output, leaving *filename* untouched until the temporary file is successfully closed. Specifically, OPEN-OUTPUT-FILE deletes the BAK file for *filename* if it should exist, creates a new directory entry for the temporary file, opens the temporary file for output and makes it the SOF, initializes its write position to 0, sets the control variable *OUTPUT-FILE* to *filename* (thus making the temporary file the SOS as well as the SOF), and returns *filename*. Note that the original file is not renamed as a BAK file until the temporary file is successfully closed (see CLOSE-OUTPUT-FILE). This automatic single level file backup feature prevents losing the original file should a disk error occur while writing to the new file.

If OPEN-OUTPUT-FILE attempts to create a new directory entry and there is no MS-DOS directory entries available, OPEN-OUTPUT-FILE returns NIL.

Since PRN is the MS-DOS device name for the system printer, (OPEN-OUTPUT-FILE 'PRN) makes the printer the SOS. This makes it possible to direct output to the printer under program control. To produce a "hard copy" record of a muLISP session, use the control variable *PRINTER ECHO* (see Section 5.14).

```
(DEFUN OPEN-OUTPUT-FILE (FILE-NAME OVERWRITE)
  (CLOSE-OUTPUT-FILE FILE-NAME)
  ( ((NULL OVERWRITE)
      (delete BAK file for FILE-NAME if it exists)
      (create a temporary file for FILE-NAME) )
    ((PROBE-FILE FILE-NAME))
    (create a new file for FILE-NAME) )
  (open FILE-NAME for output)
  (OUTPUT-FILE FILE-NAME)
  (FILE-WRITE-POSITION 0)
  (SETQ *OUTPUT-FILE* (OUTPUT-FILE)) )

$ (OPEN-OUTPUT-FILE TEST.LSP)     ;Make TEST.LSP the SOF
C:TEST.LSP

$ *OUTPUT-FILE*                   ;*OUTPUT-FILE* is now nonNIL
C:TEST.LSP

$ (FILE-WRITE-POSITION)                    ;Initial write position
0

$ (WRITE-LINE '|This is a test.|)
|This is a test.|

$ (FILE-WRITE-POSITION)                    ;Current write position
16
```

CLOSE-OUTPUT-FILE [*filename*] Function

If *overwrite* was nonNIL when *filename* was opened by OPEN-OUTPUT-FILE, (CLOSE-OUTPUT-FILE *filename*) closes *filename* for output, sets the length of the file to the current write position of *filename*, and returns T. If *filename* is also the SOF, CLOSE-OUTPUT-FILE sets the control variable *OUTPUT-FILE* to NIL.

If *overwrite* was NIL when *filename* was opened by OPEN-OUTPUT-FILE, (CLOSE-OUTPUT-FILE *filename*) closes the temporary file created for *filename*, sets the length of the temporary file to the current write position of *filename*, renames the existing *filename* to be a BAK file, renames the temporary file to *filename*, and returns T. If *filename* is also the SOF, CLOSE-OUTPUT-FILE sets the control variable *OUTPUT-FILE* to NIL.

If *filename* is not an open output file, (CLOSE-OUTPUT-FILE *filename*) returns NIL.

If there is a SOF, (CLOSE-OUTPUT-FILE) closes the SOF as described above, and returns T. If there is no SOF, (CLOSE-OUTPUT-FILE) returns NIL.

```
(DEFUN CLOSE-OUTPUT-FILE (FILE-NAME)
  ((NULL FILE-NAME)
    ((OUTPUT-FILE)
      (CLOSE-OUTPUT-FILE (OUTPUT-FILE)) ) )
  ((OUTPUT-FILE-P FILE-NAME)
    (IF (EQ FILE-NAME (OUTPUT-FILE))
        (SETQ *OUTPUT-FILE*) )
    (close FILE-NAME for output)
    (set the file length to write position)
    ((if FILE-NAME is an overwrite output file))
    (rename FILE-NAME to a BAK file)
    (rename the temporary file to FILE-NAME)
    T ) )

$ (CLOSE-OUTPUT-FILE)                    ;Close the SOF
T
```

OUTPUT-FILE [*filename*] Function

If *filename* is an open output file, (OUTPUT-FILE *filename*) makes *filename* the SOF and returns its pathname. If *filename* is not an open output file, (OUTPUT-FILE *filename*) returns NIL.

If there is a SOF, (OUTPUT-FILE) returns the pathname of the SOF. If there is no SOF, (OUTPUT-FILE) returns NIL.

```
(DEFUN OUTPUT-FILE (FILE-NAME)
  ((NULL FILE-NAME)
    ((there is a SOF)
      (the pathname of the SOF) ) )
  ((OUTPUT-FILE-P FILE-NAME)
    (make FILE-NAME the SOF)
    (the pathname of FILE-NAME) ) )
```

```
$ (OPEN-OUTPUT-FILE 'TEST1.LSP)          ;Open TEST1.LSP
C:TEST1.LSP

$ (OPEN-OUTPUT-FILE 'TEST2.LSP)          ;Open TEST2.LSP
C:TEST2.LSP

$ (OUTPUT-FILE)                          ;TEST2.LSP is the SOF
C:TEST2.LSP

$ (OUTPUT-FILE 'TEST1.LSP)               ;Make TEST1.LSP the SOF
C:TEST1.LSP

$ (OUTPUT-FILE)                          ;TEST1.LSP is the SOF
C:TEST1.LSP
```

OUTPUT-FILE-P [*filename*] Function
OUTPUT-FILES [] Function

If *filename* is an open output file, (OUTPUT-FILE-P *filename*) returns T, otherwise it returns NIL.

(OUTPUT-FILES) returns a list of the pathnames of the currently open output files.

```
(DEFUN OUTPUT-FILE-P (FILE-NAME)
  ((if FILE-NAME is an open output file) T)
  NIL )

(DEFUN OUTPUT-FILES ()
  (a list of the open output files) )

$ (OPEN-OUTPUT-FILE 'TEST1.LSP)          ;Open TEST1.LSP
C:TEST1.LSP

$ (OPEN-OUTPUT-FILE 'TEST2.LSP)          ;Open TEST2.LSP
C:TEST2.LSP

$ (OUTPUT-FILE-P 'TEST1.LSP)             ;Open for output?
T                                                ;Yes

$ (OUTPUT-FILE-P 'TEST2.LSP)             ;Open for output?
T                                                ;Yes

$ (OUTPUT-FILES)                  ;Currently open output files
(C:TEST2.LSP C:TEST1.LSP)
```

OUTPUT-FILE: NIL Control variable

If there is a SOF and *OUTPUT-FILE* is nonNIL, standard output goes to the SOF; otherwise standard output goes to the console.

Normally the functions OPEN-OUTPUT-FILE and OUTPUT-FILE provide sufficient control over the destination of character output from muLISP programs. However, after writing part of a file, it may be necessary to temporarily send output to the console before returning to the file. The control variable *OUTPUT-FILE* provides this capability.

After using OPEN-OUTPUT-FILE to open a file and make it the SOF, the console can be made the SOS without changing the SOF by setting *OUTPUT-FILE* to NIL. A subsequent nonNIL assignment to *OUTPUT-FILE* will again make the SOF the SOS, and writing of the file will resume from the point at which it had been suspended.

An alternative method of temporarily sending output to the console is to specify the console in calls on the output functions (see Section 5.14).

FILE-WRITE-POSITION [*n*, *filename*] Function

If *filename* is an open output file and *n* is zero or a positive integer, (FILE-WRITE-POSITION *n filename*) sets the write position of *filename* to *n* and returns T. If *n* is neither zero nor a positive integer, FILE-WRITE-POSITION returns the current write position of *filename*. If *filename* is not an open output file, FILE-WRITE-POSITION returns NIL. If no *filename* argument is given, the SOF is assumed.

A file's write position determines the point in the file at which the next character will be written. When a file is opened for output, the file's write position is set to 0 (the first byte of the file). After a character is written to the file, the file's write position is automatically incremented. Thus, use FILE-WRITE-POSITION when a file needs to be written to non-sequentially or when it is necessary to reduce the size of a file (see CLOSE-OUTPUT-FILE). To save the entirety of a non-sequentially written file, remember to set the file's write position to the end of the file *before* closing the file.

FILE-WRITE-POSITION makes it possible to create files containing "holes" where no characters have been written. An attempt to read characters from such holes will cause an "End-Of-File" error break.

```
(DEFUN FILE-WRITE-POSITION (N FILE-NAME)
  ((NULL FILE-NAME)
    ((OUTPUT-FILE)
      (FILE-WRITE-POSITION N (OUTPUT-FILE)) ) )
  ((OUTPUT-FILE-P FILE-NAME)
    ((AND (INTEGERP N) (>= N 0))
      (set the write position of FILE-NAME to N) )
    (FILE-NAME's current write position) ) )

$ (OPEN-OUTPUT-FILE 'TEST1.LSP)          ;Open TEST1.LSP
C:TEST1.LSP

$ (FILE-WRITE-POSITION)                  ;Initial write position
0

$ (FILE-WRITE-POSITION 100)              ;Set write position to 100
```

FILE-LINE-LENGTH [*n*, *filename*] Function

If *filename* is an open output file and *n* is a positive integer, (FILE-LINE-LENGTH *n* *filename*) sets the line length of *filename* to *n* and returns T. If *n* is not a positive integer, FILE-LINE-LENGTH returns the current line length of *filename*. If *filename* is not an open output file, FILE-LINE-LENGTH returns NIL. If no *filename* argument is given, the SOF is assumed.

If the control variable *AUTO-NEWLINE* is nonNIL, newline characters are automatically sent to an open output file as necessary to enforce the file's line length. If *AUTO-NEWLINE* is NIL, the file's line length is ignored (see *AUTO-NEWLINE* in Section 5.14).

```
(DEFUN FILE-LINE-LENGTH (N FILE-NAME)
  ((NULL FILE-NAME)
    ((OUTPUT-FILE)
      (FILE-LINE-LENGTH N (OUTPUT-FILE)) ) )
  ((OUTPUT-FILE-P FILE-NAME)
    ((AND (INTEGERP N) (PLUSP N))
      (set the line length of FILE-NAME to N)
      T )
    (FILE-NAME's current line length) ) )

$ (OPEN-OUTPUT-FILE 'TEST1.LSP)          ;Open TEST1.LSP
C:TEST1.LSP

$ (FILE-LINE-LENGTH)                     ;Default line length
79

$ (FILE-LINE-LENGTH 120)                 ;Set line length to 120
T
```

5.15.3 File and Directory Functions

The file operation functions enable programs to delete a file, rename a file, and determine a file's existence, date, and length. The directory functions enable programs to determine and/or modify the default drive and current directory, and produce a list of the files in a specified directory.

RENAME-FILE [*old-filename*, *new-filename*] Function
DELETE-FILE [*filename*] Function
PROBE-FILE [*filename*] Function

If the file on the drive and directory specified by *old-filename* exists, (RENAME-FILE *old-filename new-filename*) renames *old-filename* to *new-filename* and returns T. If no file specified by *old-filename* exists, RENAME-FILE returns NIL.

If the file on the drive and directory specified by *filename* exists, (DELETE-FILE *filename*) deletes *filename* and returns T. If no file specified by *filename* exists, DELETE-FILE returns NIL.

If the file on the drive and directory specified by *filename* exists, (PROBE-FILE *filename*) returns T. If no file specified by *filename* exists, PROBE-FILE returns NIL.

```
$ (RENAME-FILE 'TEMP.LSP 'TEST.LSP)          ;Rename file
T

$ (PROBE-FILE 'TEST.LSP)                     ;There is a TEST.LSP
T

$ (DELETE-FILE 'TEST.LSP)                    ;Delete TEST.LSP
T

$ (PROBE-FILE 'TEST.LSP)                     ;There is no TEST.LSP
NIL
```

FILE-DATE [*filename*] Function
FILE-LENGTH [*filename*] Function

If the file on the drive and directory specified by *filename* exists, (FILE-DATE *filename*) returns the date *filename* was written as a list of six integers signifying the following: year (A.D.), month (1-12), day (1-31), hour (0-23), minute (0-59), and second (0-59). If no file specified by *filename* exists, FILE-DATE returns NIL.

If *filename* is an open input and/or output file, (FILE-LENGTH *filename*) returns the length of *filename* in bytes. If *filename* is not open, FILE-LENGTH returns NIL.

```
$ (FILE-DATE 'ANIMAL.LSP)                ;Date of ANIMAL.LSP
(1986 12 23 10 43 4)                     ;December 23, 1986, 10:43:04

$ (OPEN-INPUT-FILE 'ANIMAL.LSP)          ;Open ANIMAL.LSP
C:ANIMAL.LSP

$ (FILE-LENGTH 'ANIMAL.LSP)              ;Length of ANIMAL.LSP
5923
```

DEFAULT-DRIVE [*drive*] Function
DEFAULT-PATH [*drive*, *path*] Function
DIRECTORY [*pathname*] Function

If *drive* is a valid drive letter, (DEFAULT-DRIVE *drive*) sets the MS-DOS default drive to *drive* and returns the total number of drives. (DEFAULT-DRIVE) returns the drive letter of the default drive.

If *drive* is a valid drive letter and path is a valid directory path, (DEFAULT-PATH *drive path*) sets the MS-DOS current directory on *drive* to *path* and returns T. (DEFAULT-PATH *drive*) returns the current directory on *drive*. (DEFAULT-PATH) returns the current directory on the default drive.

If *pathname* specifies a valid drive and directory, (DIRECTORY *pathname*) returns a list of the names of the files in that directory that match *pathname*. The MS-DOS wild-card filename characters, "?" and "*", can be used to specify multiple files.

```
$ (DEFAULT-DRIVE)                          ;Default drive is A
A

$ (DEFAULT-DRIVE 'C)                       ;Change drive to C
2                                          ;A 2 drive system

$ (DEFAULT-PATH)                    ;Current directory is MULISP
MULISP

$ (DEFAULT-PATH 'C 'BIN)                   ;Change directory to C:\BIN
T

$ (DIRECTORY '|*.COM|)          ;The COM files in current directory
(MULISP.COM COMTOEXE.COM)
```

5.15.4 Environment Functions

The current muLISP environment consists of all the variable values, function definitions, and property values presently in the system. These values and definitions could have been entered directly from the console, read from a source file, or loaded from a SYS file. The environment functions make it possible to load and save the muLISP environment as a compact, memory image SYS file. A SYS file is usually created only after a program has been thoroughly debugged. Loading a SYS file is usually much faster than reading in the equivalent source file(s).

SAVE [*filename*] Function

If *filename* specifies a file name extension *other* than COM, (SAVE *filename*) saves the current muLISP environment as a SYS file on the drive and directory specified by *filename* and returns T. SAVE always gives SYS files a file name extension of SYS. If there is no MS-DOS directory entries available or there is insufficient disk space for *filename*, SAVE returns NIL.

If *filename* specifies a file name extension of COM, (SAVE *filename*) saves the current muLISP environment and the muLISP interpreter, minus the pseudo-code compiler, as an executable COM file on the drive and directory specified by *filename*, and returns T. By modifying a flag in the muLISP Base Page, it is possible to retain the pseudo-code compiler in COM files created by SAVE (see Chapter 9 and Appendix B).

Prior to saving an environment as a SYS or COM file, SAVE automatically compacts all active data structures so that the resulting file is of minimum size. Therefore, the size of the memory image file increases with the amount of active data structures in the muLISP environment at the time of the save.

If the COM file produced by SAVE is less than 65280 bytes, it can be executed as an MS-DOS command. If the COM file is greater than or equal to 65280 bytes, it must first be converted to an EXE file using the utility COMTOEXE.COM distributed with muLISP. If the COM file specified by *filename* exists, the MS-DOS command

COMTOEXE *filename drive*

creates an EXE file on *drive* equivalent to the COM file and then deletes the COM file. If *drive* is omitted, the EXE file is created on the MS-DOS default drive.

Since COM files are larger than SYS files, have their pseudo-code compiler disabled, and cannot be loaded into a running muLISP system, SYS files are normally used during program development. COM files, on the other hand, are primarily useful for packaging completed application programs.

If an application program is saved as a COM or EXE file with the pseudo-code compiler disabled so that the end user cannot define additional muLISP functions, the COM or EXE file can be sold without requiring the end-users to pay for a full muLISP system. Instead, the application program developer can pay a small royalty for the use of the muLISP runtime system contained in the COM or EXE file. To obtain a muLISP Runtime License Agreement for the rights to distribute the runtime system with your application program, contact Soft Warehouse, Inc., 3615 Harding Avenue, Suite 505, Honolulu, Hawaii, 96816, U.S.A.

```
(DEFUN SAVE (FILE-NAME)
  (close all open input and output files)
  (RECLAIM)
  ((FILE-NAME specifies a COM file)
    (disable the muLISP pseudo-code compiler)
    (save the muLISP interpreter and the current
        environment as a COM file)
    T )
  (save the current environment as a SYS file)
  T )

$ (SAVE 'B:WHALE)                    ;Save WHALE.SYS on drive B:
T

$ (SAVE 'WHALE.COM)                  ;Save WHALE.COM on default drive
T
```

LOAD [*filename*, *display*] — Function
LOAD-VERBOSE: T — Control variable

If *filename* specifies a file name extension of SYS and the file on the drive and directory specified by *filename* exists, (LOAD *filename*) *replaces* the current muLISP environment with the environment present at the time the SYS file *filename* was created. LOAD does *not* return a value if the SYS file is successfully loaded, but begins execution of the function bound to the variable DRIVER in the new environment.

If *filename* specifies a file name extension *other* than SYS and the file on the drive and directory specified by *filename* exists, (LOAD *filename* *display*) successively reads in and evaluates the expressions in the source file *filename* and returns T. If *LOAD-VERBOSE* is nonNIL, LOAD displays the message ";Loading *filename*". If *display* is nonNIL, LOAD displays the value of each expression after it is evaluated. This facilitates finding syntax errors in large source files. Examining the value of the variables +, ++, +++, *, **, and *** may also be useful (see Section 5.19).

If no file name extension is specified in *filename*, LOAD assumes an extension name of SYS. If such a SYS file exists, it is loaded as described above. If no SYS file exists, LOAD assumes an extension name of LSP. If such a source file exists, it is read in and evaluated as described above. If no file specified by *filename* exists, LOAD returns NIL.

```
(DEFUN LOAD (FILE-NAME DISPLAY)
  ((FILE-NAME specifies no file name extension)
    ((LOAD FILE-NAME.SYS DISPLAY))
    (LOAD FILE-NAME.LSP DISPLAY) )
  ((FILE-NAME specifies a SYS file)
    (close all open input and output files)
    ((the SYS file FILE-NAME exists)
      (load and execute FILE-NAME) ) )
  ((the source file FILE-NAME exists)
    (read in and evaluate FILE-NAME)  T ) )

$ (LOAD 'C:WHALE)              ;Load WHALE.SYS from drive C

$ (LOAD 'COMMON)               ;Read in and evaluate COMMON.LSP
T
```

5.16 EVALUATION FUNCTIONS

Normally expression evaluation and function invocation are performed automatically by the muLISP interpreter. The evaluation functions are for those occasions where it is necessary to explicitly evaluate an expression or invoke a function. A function definition is a list composed of three parts: the function type name, the formal arguments, and the function body.

The car of a function definition is the function type name, either LAMBDA, NLAMBDA, or MACRO. A function's type informs the interpreter how applications of the function are to proceed.

A function of type LAMBDA is called an eval function. When an eval function is called, each of the arguments are evaluated and only the values of the arguments are passed to the function. Eval functions are by far the most common type of function used in LISP programs.

A function of type NLAMBDA is called a no-eval function. When a no-eval function is called, the arguments are passed to the function without evaluation just as they appear in the call. Since no-eval functions can lead to variable name conflicts, it is usually preferable to define a macro function rather than a no-eval function.

A function of type MACRO is called a macro function. When a macro function is called, the macro definition is first applied to the entire macro call without evaluation. The result of this macro expansion is then evaluated and returned as the value of the macro. The variable MACROEXPAND controls when macros are expanded.

The cadr of a function definition is either the formal argument list or formal argument name. A function defined using a list of formal arguments, possibly empty, is called a spread function. When a spread function is called, the actual arguments are bound to (i.e. spread over) each element of the formal argument list.

A function defined using a formal argument name, rather than a formal argument list, is called a no-spread function. When a no-spread function is called, a list of the actual arguments is bound to the formal argument name. Thus no-spread functions can accept an arbitrary number of actual arguments. Note that in muLISP the decision to make a function spread or no-spread is made independently of the decision to make it eval or no-eval.

The cddr of a function definition is a list of forms called the function body. After a function's formal arguments have been bound to the actual arguments, an implicit PROGN is used to evaluate the function body (see Section 5.17). Upon completion of the function body evaluation, the formal arguments are restored to their original value, and the value of the function body evaluation is returned as the value of the function.

5.16.1 Evaluation Recognizers

The recognizers in this subsection can be used to determine a symbol's function type and an object's characteristics if evaluated as a form.

EVAL-FUNCTION-P [*symbol*] Library Function
NO-EVAL-FUNCTION-P [*symbol*] Library Function
MACRO-FUNCTION-P [*symbol*] Library Function
SPECIAL-FORM-P [*symbol*] Library Function
FBOUNDP [*symbol*] Library Function

These predicates examine *symbol*'s function cell and return T if *symbol* is a function of the given type, otherwise they return NIL. EVAL-FUNCTION-P tests for eval functions. NO-EVAL-FUNCTION-P tests for no-eval functions. MACRO-FUNCTION-P tests for macro functions. SPECIAL-FORM-P tests for special forms. FBOUNDP tests for functions of any of these types.

```
(DEFUN EVAL-FUNCTION-P (SYM)
  (EQ (GETD SYM T) 'LAMBDA) )

(DEFUN NO-EVAL-FUNCTION-P (SYM)
  (EQ (GETD SYM T) 'NLAMBDA) )

(DEFUN MACRO-FUNCTION-P (SYM)
  (EQ (GETD SYM T) 'MACRO) )

(DEFUN SPECIAL-FORM-P (SYM)
  (EQ (GETD SYM T) 'SPECIAL) )

(DEFUN FBOUNDP (SYM)
  (GETD SYM T) )

$ (EVAL-FUNCTION-P 'APPEND)          ;APPEND is a function
T

$ (MACRO-FUNCTION-P 'DO)                    ;DO is a macro
T

$ (SPECIAL-FORM-P 'IF)                    ;IF is a special form
T

$ (FBOUNDP NIL)                  ;NIL is not a defined function
NIL
```

CONSTANTP [*object*] Function
FUNCTIONP [*object*] Library Function

If *object* is NIL, a number, or a list whose car is the symbol QUOTE, (CONSTANTP *object*) returns T, otherwise it returns NIL. If *object* is a constant as determined by CONSTANTP, *object* always returns the same value when evaluated as a form.

If *object* is a symbol or a list whose car is the symbol LAMBDA, (FUNCTIONP *object*) returns T, otherwise it returns NIL. If *object* is a function as determined by FUNCTIONP, *object* is suitable for use as the functional argument to APPLY, FUNCALL, and the mapping functions.

```
(DEFUN CONSTANTP (OBJ)
  ((OR (NULL OBJ) (NUMBERP OBJ)))
  ((ATOM OBJ) NIL)
  (EQ (CAR OBJ) 'QUOTE) )

(DEFUN FUNCTIONP (OBJ)
  ((SYMBOLP OBJ))
  ((ATOM OBJ) NIL)
  (EQ (CAR OBJ) 'LAMBDA) )

$ (CONSTANTP ())                          ;NIL is a constant
T

$ (CONSTANTP -237.6)                      ;Numbers are constants
T

$ (CONSTANTP '(QUOTE (A B C)))            ;Quotes are constant
T

$ (FUNCTIONP 'FOO)                        ;FOO is a suitable function
T

$ (FUNCTIONP '(+ 2 3))                    ;Lists are not functions
NIL

$ (FUNCTIONP '(LAMBDA (X Y) (- Y X)))
T                                         ;Lambdas are suitable functions
```

5.16.2 Evaluation and Function Application

The primitive functions in this subsection make it possible to explicitly evaluate expressions and to apply functions to arguments.

SYMBOL-VALUE [*symbol*] Library Function
SYMBOL-PLIST [*symbol*] Library Function
SYMBOL-FUNCTION [*symbol*] Library Function

(SYMBOL-VALUE *symbol*) returns *symbol*'s current value (i.e. the object pointed to by *symbol*'s value cell).

(SYMBOL-PLIST *symbol*) returns *symbol*'s current property list (i.e. the object pointed to by *symbol*'s property list cell).

(SYMBOL-FUNCTION *symbol*) returns *symbol*'s current function definition (i.e. the object or address pointed to by *symbol*'s definition cell). If *symbol* is undefined (see FBOUNDP), SYMBOL-FUNCTION generates an "Undefined Function" error break.

```
(DEFUN SYMBOL-VALUE (SYM)
  ((SYMBOLP SYM)
    (CAR SYM) ) )

(DEFUN SYMBOL-PLIST (SYM)
  ((SYMBOLP SYM)
    (CDR SYM) ) )

(DEFUN SYMBOL-FUNCTION (SYM)
  ((SYMBOLP SYM)
    ((FBOUNDP SYM)
      (GETD SYM) )
    (BREAK (LIST 'SYMBOL-FUNCTION SYM)
           "Undefined Function") ) )

$ (SYMBOL-FUNCTION 'FBOUNDP)          ;FBOUNDP's definition
(LAMBDA (SYM) (GETD SYM T))
```

EVAL [*object*] Function

If *object* is an atom, (EVAL *object*) returns the content of *object*'s value cell. Since new atoms are auto-quoted when they are interned, they evaluate to themselves until they are bound to another value.

If the car of *object* is the name of an eval function or a LAMBDA body, (EVAL *object*) evaluates each element of the cdr of *object* and applies the car of *object* to a list of the results.

If the car of *object* is the name of a no-eval function, (EVAL *object*) applies the car of *object* to the cdr of *object* without evaluation of the latter.

If the car of *object* is the name of a macro function, (EVAL *object*) recursively evaluates the result of applying the car of *object* to *object*.

If the car of *object* is not a function, (EVAL *object*) calls the undefined function error handler UNDEFINED which in turn generates an "Undefined Function" error break.

```
(DEFUN EVAL (OBJ)
  ((ATOM OBJ)
    (CAR OBJ) )
  ((SYMBOLP (CAR OBJ))
    ((EVAL-FUNCTION-P (CAR OBJ))
      (APPLY (CAR OBJ) (MAPCAR 'EVAL (CDR OBJ))) )
    ((NO-EVAL-FUNCTION-P (CAR OBJ))
      (APPLY (CAR OBJ) (CDR OBJ)) )
    ((MACRO-FUNCTION-P (CAR OBJ))
      (EVAL (APPLY (CAR OBJ) OBJ)) )
    (APPLY 'UNDEFINED OBJ) )
  ((FUNCTIONP (CAR OBJ))
    (APPLY (CAR OBJ) (MAPCAR 'EVAL (CDR OBJ))) )
  (APPLY 'UNDEFINED OBJ) )
```

```
$ (SETQ JOHN 'MAN)                    ;Set value of JOHN to MAN
MAN

$ (SETQ MAN 'ANIMAL)                  ;Set value of MAN to ANIMAL
ANIMAL

$ JOHN                                ;Value of JOHN is MAN
MAN

$ (EVAL JOHN)                         ;Value of value of JOHN is ANIMAL
ANIMAL

$ (EVAL '(CONS 'A '(B C D)))          ;Value of a form
(A B C D)

$ ((LAMBDA (N) (* N N)) 5)            ;Call an in-line LAMBDA
25
```

APPLY [*function*, arg_1, arg_2, ..., *arglist*] Function

If *function* is the name of a machine language routine, (APPLY *function* arg_1 arg_2 ... *arglist*) passes the actual arguments (i.e. arg_1, arg_2, ... plus the elements of *arglist*) to the machine language routine, and returns the result of the routine.

If *function* is the name of a user-defined function or a LAMBDA body, APPLY temporarily binds the formal arguments of *function* to the actual arguments (i.e. arg_1, arg_2, ... plus the elements of *arglist*), evaluates the function body, restores the original values of the formal arguments, and returns the value of the function body evaluation.

If *function* is not the name of a function or a LAMBDA body, APPLY generates an "Undefined Function" error break.

```
(DEFUN APPLY (FUNC ARG1 ARG2 ... ARGS)
  (SETQ ARGS (LIST* ARG1 ARG2 ... ARGS))
  ((SYMBOLP FUNC)
    ((machine language routine FUNC)
      (execute the machine language routine using
          ARGS for the arguments) )
    ((OR (EVAL-FUNCTION-P FUNC)
         (NO-EVAL-FUNCTION-P FUNC))
      (eval-body ARGS (arguments FUNC)
                 (body FUNC)) )
    (APPLY 'UNDEFINED (CONS FUNC ARGS)) )
  ((OR (EQ (CAR FUNC) 'LAMBDA)
       (EQ (CAR FUNC) 'NLAMBDA))
    (eval-body ARGS (CADR FUNC) (CDDR FUNC)) )
  (APPLY 'UNDEFINED (CONS FUNC ARGS)) )
```

```
(DEFUN eval-body (ACTUAL FORMAL BODY)
  ((LISTP FORMAL)
    (MAPC '(LAMBDA (X) (PUSH (EVAL X) STACKLIST))
          FORMAL)
    (MAPC 'SET FORMAL ACTUAL)
    (PROG1 (PROGN BODY)
       (MAPC '(LAMBDA (X) (SET X (POP STACKLIST)))
             FORMAL)) )
  (PUSH (EVAL FORMAL) STACKLIST)
  (SET FORMAL ACTUAL)
  (PROG1 (PROGN BODY)
         (SET FORMAL (POP STACKLIST))) )

$ (APPLY 'CONS '(A (B C D)))                ;Apply CONS
(A B C D)

$ (APPLY 'CONS 'A '((B C D)))               ;Apply CONS
(A B C D)

$ (APPLY 'CONS 'A '(B C D) NIL)             ;Apply CONS
(A B C D)

$ (APPLY '(LAMBDA (N) (* N N)) '(5))  ;Apply LAMBDA
25
```

FUNCALL [*function*, arg_1, arg_2, ..., arg_n] Function

(FUNCALL *function* arg_1 arg_2 ... arg_n) applies *function* to the argument arg_1, arg_2, ..., arg_n, and returns the result. *function* must either be the name of an eval or no-eval function, or a LAMBDA body. If *function* is the name of a macro or an undefined function, an "Undefined Function" error break occurs.

```
(DEFUN FUNCALL (FUNC ARG1 ARG2 ... ARGn)
  (APPLY FUNC ARG1 ARG2 ... ARGn NIL) )

$ (FUNCALL 'CONS 'A '(B C D))                ;Apply CONS
(A B C D)

$ (FUNCALL '(LAMBDA (N) (* N N)) 5)    ;Apply LAMBDA
25
```

INTERRUPT-HOOK: 'CONSOLE-BREAK — Control variable
CONSOLE-BREAK [] — Function
UNDEFINED [*symbol*, *form*$_1$, ..., *form*$_n$] — Function

If *INTERRUPT-HOOK* is the name of a function and the ESC key is pressed during the execution of a program, the program is suspended while the function is called with no arguments. If *INTERRUPT-HOOK* is NIL, user initiated console interrupts are disabled and the ESC key is read just like any other character that has been entered from the keyboard. Otherwise, an "Undefined Function" error break occurs. *INTERRUPT-HOOK* makes it possible for a program to customize the handling of console interrupts appropriate for a given application.

(CONSOLE-BREAK) generates a console interrupt break by evaluating the form (BREAK NIL "Console Interrupt"). CONSOLE-BREAK is the initial value of *INTERRUPT-HOOK*. See the discussion of BREAK in Section 5.19 for the options available during a break.

(UNDEFINED *symbol* *form*$_1$ ... *form*$_n$) generates an "Undefined Function" error break by evaluating the form (BREAK (LIST *symbol* *form*$_1$ ... *form*$_n$) "Undefined Function"). This error handler is called when an attempt is made to evaluate a form whose car is a symbol having no function definition. The errant form and the message "Undefined Function" are passed to BREAK to help the user diagnose the error. See the discussion of BREAK in Section 5.19 for the options available during a break.

```
(DEFUN CONSOLE-BREAK ()
  (BREAK NIL "Console Interrupt") )

(DEFUN UNDEFINED LST
  (BREAK LST "Undefined Function") )
```

5.16.3 Macro Expansion Functions

A macro call is a list of the form (*macro form*$_1$ *form*$_2$ ... *form*$_n$) where *macro* is the name of a macro function. Normally macro expansion occurs automatically when a macro call is pseudo-compiled or run. However, the functions in this subsection allow the user to explicitly expand macro calls.

MACRO-FUNCTION [*symbol*] Library Function

If *symbol* is defined as a macro, (MACRO-FUNCTION *symbol*) returns the *symbol*'s macro definition. Otherwise MACRO-FUNCTION returns NIL.

```
(DEFUN MACRO-FUNCTION (SYM)
  ((MACRO-FUNCTION-P SYM)
    (GETD SYM) ) )
```

```
$ (MACRO-FUNCTION 'WHEN)          ;WHEN's macro definition
(MACRO BODY ((LAMBDA (TEST BODY)
(LIST (QUOTE IF) TEST (CONS (QUOTE PROGN) BODY)))
(CAR (CDR BODY))
(CDR (CDR BODY))))

$ (MACRO-FUNCTION 'FOO)                ;FOO is not a macro
NIL
```

MACROEXPAND-1 [*form*] Function
MACROEXPAND [*form*] Function

If *form* is a macro call, (MACROEXPAND-1 *form*) expands *form* (i.e. applies the macro function to *form*) and returns the result. Otherwise, MACROEXPAND-1 returns *form*.

If *form* is a macro call, (MACROEXPAND *form*) repeatedly expands *form* until the result of the expansion is not a macro call. MACROEXPAND returns the result of the last expansion.

```
(DEFUN MACROEXPAND-1 (FORM)
  ((ATOM FORM) FORM)
  ((NOT (SYMBOLP (CAR FORM))) FORM)
  ((NOT (EQ 'MACRO (GETD (CAR FORM) T))) FORM)
  (APPLY (CAR FORM) FORM) )

(DEFUN MACROEXPAND (FORM)
  ((ATOM FORM) FORM)
  ((NOT (SYMBOLP (CAR FORM))) FORM)
  ((NOT (EQ 'MACRO (GETD (CAR FORM) T))) FORM)
  (MACROEXPAND (APPLY (CAR FORM) FORM)) )

$ (DEFMACRO EQCAR (OBJ SYM)          ;Define EQCAR macro
    (LIST 'EQ (LIST 'CAR OBJ) SYM) )
EQCAR

$ (EQCAR '(+ 2 3 5) '+)                    ;Test EQCAR macro
T

$ (MACROEXPAND '(EQCAR '(+ 2 3 5) '+))
(EQ (CAR (QUOTE (+ 2 3 5))) (QUOTE +)) ;Macro expansion
```

MACROEXPAND: T — Control variable

If MACROEXPAND is nonNIL, macro calls are expanded as if by the function MACROEXPAND at compile time (i.e. when a function is defined using DEFUN or PUTD). If MACROEXPAND is NIL, macro calls are not expanded at compile time.

Although muLISP automatically expands macro calls at run time (i.e. when a function is being evaluated), it is usually more efficient to expand macros just once at compile time. For macro expansion to occur at compile time, the macro must be defined *before* the function containing the macro call is defined, and MACROEXPAND must be nonNIL.

Since the muLISP pseudo-code compiler is non-reentrant, macro calls that define functions must not be expanded at compile time. Such function defining macros can be expanded at run time.

```
$ (SETQ MACROEXPAND)                ;Turn off compile time expansion

$ (DEFUN IS-SUM (EXPN)                ;Define IS-SUM using EQCAR
    (EQCAR EXPN '+) )                 ;macro defined on previous page
IS-SUM

$ (GETD IS-SUM)                                ;Definition of IS-SUM
(LAMBDA (EXPN) (EQCAR EXPN (QUOTE +)))

$ (IS-SUM '(+ 2 3 5))                          ;Verify IS-SUM working
T

$ (SETQ MACROEXPAND T)              ;Turn on compile time expansion

$ (DEFUN IS-SUM (EXPN)                         ;Redefine IS-SUM
    (EQCAR EXPN '+) )
IS-SUM

$ (GETD IS-SUM)                             ;New definition of IS-SUM
(LAMBDA (EXPN) (EQ (CAR EXPN) (QUOTE +)))

$ (IS-SUM '(+ 2 3 5))                       ;IS-SUM is still working
T                                                  ;and it is faster
```

5.16.4 Function Mapping Functions

The functions in this subsection successively apply a function to the elements or sublists of one or more lists. The different mapping functions provide alternatives for returning the results of the function applications.

MAPC [*function*, $list_1$, $list_2$, ..., $list_n$] Function
MAPL [*function*, $list_1$, $list_2$, ..., $list_n$] Function

(MAPC *function* $list_1$ $list_2$ ... $list_n$) applies *function* to the car of $list_1$ through $list_n$, then to the cadr of each list, and so on until any of the lists are exhausted. MAPC returns $list_1$.

(MAPL *function* $list_1$ $list_2$ ... $list_n$) applies *function* to $list_1$ through $list_n$, then to the cdr of each list, and so on until any of the lists are exhausted. MAPL returns $list_1$.

```
(DEFUN MAPC (FUNC LST1 LST2)
  ((OR (NULL LST1) (NULL LST2)) LST1)
  (FUNCALL FUNC (CAR LST1) (CAR LST2))
  (MAPC FUNC (CDR LST1) (CDR LST2))
  LST1 )

(DEFUN MAPL (FUNC LST1 LST2)
  ((OR (NULL LST1) (NULL LST2)) LST1)
  (FUNCALL FUNC LST1 LST2)
  (MAPL FUNC (CDR LST1) (CDR LST2))
  LST1 )

$ (MAPC 'PRIN1 '(DOG CAT COW))          ;Print animals
DOGCATCOW(DOG CAT COW)                  ;and return list

$ (MAPL '(LAMBDA (LST) (RPLACA LST NIL))
        '(A B C D E))
(NIL NIL NIL NIL NIL)        ;Replace elements of list with NIL
```

MAPCAR [*function*, $list_1$, $list_2$, ..., $list_n$] Function
MAPLIST [*function*, $list_1$, $list_2$, ..., $list_n$] Function

(MAPCAR *function* $list_1$ $list_2$ ... $list_n$) applies *function* to the car of $list_1$ through $list_n$, then to the cadr of each list, and so on until any of the lists are exhausted. MAPCAR returns a list of the results.

(MAPLIST *function* $list_1$ $list_2$ ... $list_n$) applies *function* to $list_1$ through $list_n$, then to the cdr of each list, and so on until any of the lists are exhausted. MAPLIST returns a list of the results.

```
(DEFUN MAPCAR (FUNC LST1 LST2)
  ((OR (NULL LST1) (NULL LST2)) NIL)
  (CONS (FUNCALL FUNC (CAR LST1) (CAR LST2))
        (MAPCAR FUNC (CDR LST1) (CDR LST2))) )

(DEFUN MAPLIST (FUNC LST1 LST2)
  ((OR (NULL LST1) (NULL LST2)) NIL)
  (CONS (FUNCALL FUNC LST1 LST2)
        (MAPLIST FUNC (CDR LST1) (CDR LST2))) )

$ (MAPCAR '* '(2 3 5 7) '(2 4 6 8))
(4 12 30 56)                    ;Pairwise multiply elements of list

$ (MAPLIST 'REVERSE '(A B C D))
((D C B A) (D C B) (D C) (D))       ;Reverse list and its tails
```

MAPCAN [*function*, $list_1$, $list_2$, ..., $list_n$] Function
MAPCON [*function*, $list_1$, $list_2$, ..., $list_n$] Function

(MAPCAN *function* $list_1$ $list_2$ ... $list_n$) applies *function* to the car of $list_1$ through $list_n$, then to the cadr of each list, and so on until any of the lists are exhausted. MAPCAN concatenates the results using NCONC. MAPCAN is useful for filtering out undesired elements of a list.

(MAPCON *function* $list_1$ $list_2$ ... $list_n$) applies *function* to $list_1$ through $list_n$, then to the cdr of each list, and so on until any of the lists are exhausted. MAPCON concatenates the results using NCONC. Note that MAPCON can easily generate circular lists, so beware.

```
(DEFUN MAPCAN (FUNC LST1 LST2)
  ((LAMBDA (LST) (LOOP
    ((OR (NULL LST1) (NULL LST2)) LST)
    (SETQ LST (NCONC LST
     (FUNCALL FUNC (POP LST1) (POP LST2)))) ) )) )

(DEFUN MAPCON (FUNC LST1 LST2)
  ((LAMBDA (LST) (LOOP
    ((OR (NULL LST1) (NULL LST2)) LST)
    (SETQ LST
          (NCONC LST (FUNCALL FUNC LST1 LST2)))
    (POP LST1)
    (POP LST2) ) )) )

$ (MAPCAN '(LAMBDA (N) ((>= N 0) (LIST N)))
          '(3 -7 4 0 -5 1))
(3 4 0 1)                          ;Eliminate negative numbers

$ (MAPCON 'REVERSE '(A B C D))
(D C B A D C B D C D)              ;Concatenate reversed tails
```

5.16.5 Function Mapping Predicates

The mapping predicates in this subsection successively apply a test function to the elements of one or more lists until a termination criterion is met or the end of any of the lists is reached. The termination criterion is based on the truth value returned by the test function. The mapping predicates return a truth value based on whether or not the end of any of the lists is reached.

SOME [*test*, $list_1$, $list_2$, ..., $list_n$] Function
NOTANY [*test*, $list_1$, $list_2$, ..., $list_n$] Function

(SOME *test* $list_1$ $list_2$ ... $list_n$) applies the predicate *test* to the car of $list_1$ through $list_n$, then to the cadr of each list, and so on until a test returns a nonNIL value or the end of a list is reached. If a test returns a nonNIL value, SOME returns that value. If the end of a list is reached, SOME returns NIL.

(NOTANY *test* $list_1$ $list_2$ ... $list_n$) is equivalent to (NOT (SOME *test* $list_1$ $list_2$ ... $list_n$)).

```
(DEFUN SOME (TST LST1 LST2)
  (LOOP
    ((OR (NULL LST1) (NULL LST2)) NIL)
    ((FUNCALL TST (POP LST1) (POP LST2))) ) )

(DEFUN NOTANY (TST LST1 LST2)
  (NOT (SOME TST LST1 LST2)) )

$ (SOME 'EQL '(DOG CAT COW) '(COW CAT DOG))
T                                    ;Some pairs are EQL.

$ (SOME 'PLUSP (LIST 0 -3 (+ 4 -6)))
NIL                                  ;No element is positive

$ (NOTANY 'ODDP (LIST 0 (+ 3 3) 7/2))
T                                    ;Not any element is ODD
```

EVERY [*test*, $list_1$, $list_2$, ..., $list_n$] Function
NOTEVERY [*test*, $list_1$, $list_2$, ..., $list_n$] Function

(EVERY *test* $list_1$ $list_2$ ... $list_n$) applies the predicate *test* to the car of $list_1$ through $list_n$, then to the cadr of each list, and so on until a test returns NIL or the end of a list is reached. If a test returns NIL, EVERY returns NIL. If the end of a list is reached, it returns T.

(NOTEVERY *test* $list_1$ $list_2$ ... $list_n$) is equivalent to (NOT (EVERY *test* $list_1$ $list_2$ ... $list_n$))

```
(DEFUN EVERY (TST LST1 LST2)
  (LOOP
    ((OR (NULL LST1) (NULL LST2)))
    ((NOT (FUNCALL TST (POP LST1) (POP LST2)))
      NIL) ) )

(DEFUN NOTEVERY (TST LST1 LST2)
  (NOT (EVERY TST LST1 LST2)) )

$ (EVERY 'EQL '(DOG CAT COW) '(DOG CAT PIG))
NIL                                    ;Not every pair is EQL

$ (EVERY 'ODDP (LIST 3 5 7 11 13))
T                                      ;Every element is odd

$ (NOTEVERY 'EQL '(DOG CAT COW) '(DOG CAT PIG))
T                                      ;Not every element is EQL

$ (NOTEVERY 'STRING< '(BILL JACK JOE)
                     '(BUD JIM SUE))
NIL                          ;Every pair is in alphabetical order
```

REDUCE [*function*, *list*, *initial*] Function

If arg_1, arg_2, arg_3 ... and arg_n are the successive elements of *list* and *function* is a two argument function, (REDUCE *function list*) returns

(*function* ... (*function* (*function* arg_1 arg_2) arg_3) ... arg_n)

If *list* is an atom, (REDUCE *list function*) returns (*function*). (REDUCE *function list initial*) is equivalent to (REDUCE *function* (CONS *initial list*)), but requires one less cons.

The reduction is always left-associative. REDUCE makes it possible to apply two argument functions, like UNION and INTERSECTION, to any number of arguments.

```
(DEFUN REDUCE ARGS
  ((OR (NULL ARGS) (NULL (CDR ARGS))) NIL)
  ((NULL (CDDR ARGS))
    ((ATOM (CADR ARGS))
      (FUNCALL (CAR ARGS)) )
    (reduce-aux (CAR ARGS) (CDADR ARGS)
                (CAADR ARGS)) )
  (reduce-aux (CAR ARGS) (CADR ARGS)
              (CADDR ARGS)) )

(DEFUN reduce-aux (FUNC LST RSLT)
  ((ATOM LST) RSLT)
  (reduce-aux FUNC (CDR LST)
              (FUNCALL FUNC RSLT (CAR LST))) )

$ (REDUCE 'UNION '((A H E C) (G B A E) (C F A G)))
(H B E C F A G)                          ;Union of 3 sets

$ (REDUCE '* '(2 3 5 7))          ;Product of a list of numbers
210

$ (REDUCE '* '(2 3 5 7) -2)              ;-2 times product
-420
```

5.17 CONTROL CONSTRUCTS

Like other modern LISP systems, muLISP uses an implicit PROGN to evaluate the forms making up a function body. However, the muLISP interpreter also recognizes implicit CONDs in the function body. This control facility uses the tree structure of the function body to provide a powerful and natural conditional fork capability in definitions. The implicit COND makes muLISP function definitions more readable, concise, and efficient. A detailed description of the recognition and evaluation of implicit CONDs is given in the description of the PROGN control construct.

The special forms described in this section provide explicit control over form evaluation during program execution. Although the implicit COND greatly reduces their use in muLISP programs, explicit control constructs make possible such capabilities as iteration, non-local exits, and compatibility with other LISP systems.

QUOTE [*object*] Special form

(QUOTE *object*) returns *object* without evaluating it. *object* can be an muLISP object (i.e. either a symbol, a number, or a cons), but it is not necessary to quote NIL or numbers. QUOTE can be used to prevent the evaluation of a constant value being passed as an argument to an eval function.

The single quote mark, ', is a primitively defined read macro that can be used as a convenient abbreviation for QUOTE (see Section 5.13).

```
(DEFUN QUOTE (NLAMBDA (OBJ)
  OBJ ))

$ (SETQ DOG 'ROVER)                    ;Assign DOG a valu
ROVER

$ (UNPACK DOG)                         ;Unpack value of DOG
(R O V E R)

$ (UNPACK (QUOTE DOG))                 ;Unpack DOG itself
(D O G)

$ (CAR (CONS 4 7))                     ;Car of dotted pai
4

$ (CAR '(CONS 4 7))                    ;Car of 3 element li
CONS
```

PROGN [$form_1$, $form_2$, ..., $form_n$] Special form

(PROGN $form_1$ $form_2$ ... $form_n$) sequentially evaluates the forms beginning with $form_1$ and returns the value of the last form that is evaluated. How a form is evaluated depends on the list structure of the form as follows:

If *form* is an atom, or the car of *form* is an atom or a LAMBDA body, (EVAL *form*) is the value of *form*.

If the caar of *form* is an atom or a LAMBDA body, the car of *form* is the predicate and the cdr of *form* is the body of an implicit COND. If the value of the predicate [i.e. (EVAL (CAR *form*))] is NIL, the value of *form* is NIL. However, if the value of the predicate is nonNIL, the remaining forms of the PROGN are abandoned and the forms in the body of the implicit COND are sequentially evaluated instead. PROGN returns the value of the last form. If the body of the implicit COND is empty, PROGN returns the value of the predicate.

If *form* is none of the above, (APPLY 'PROGN *form*) is evaluated and returned as the value of *form* before continuing with the evaluation of the remaining forms. This provides for conditional forks to recombine at a later time.

```
(DEFUN PROGN (NLAMBDA FORM-LST
  (eval-progn FORM-LST) ))

(DEFUN eval-progn (LST RSLT)
  ((ATOM LST) RSLT)
  ((OR (ATOM (CAR LST)) (ATOM (CAAR LST))
       (EQ (CAAAR LST) 'LAMBDA))
    (eval-progn (CDR LST) (EVAL (CAR LST))) )
  ((OR (ATOM (CAAAR LST))
       (EQ (CAAAAR LST) 'LAMBDA))
    ((SETQ RSLT (EVAL (CAAR LST)))
      (eval-progn (CDAR LST) RSLT) )
    (eval-progn (CDR LST)) )
  (eval-progn (CDR LST) (eval-progn (CAR LST))) )
```

LOOP [$form_1$, $form_2$, ..., $form_n$] Special form

(LOOP $form_1$ $form_2$ ... $form_n$) repeatedly evaluates the forms in sequential order until an implicit COND with a nonNIL predicate is encountered. LOOP then evaluates the body of the implicit COND and returns the result.

The implicit PROGN function body evaluation mechanism is very powerful but it does not have any provisions for nonrecursive program control structures. Such iterative capability is provided by the LOOP control construct. LOOP evaluates its forms sequentially in a manner identical to the evaluation of forms in a PROGN. However, if all of a LOOP's forms are evaluated without a conditional having been satisfied, evaluation begins again with the first form.

Note that any number of predicates can occur within a loop, and they can be located at the beginning, middle, or end of the loop. This flexible loop control construct preserves the structured nature and clarity of muLISP programs while improving their performance.

The source files distributed with muLISP contain numerous examples using the LOOP control construct. Although this formal definition is recursive, LOOP is actually a nonrecursive machine language routine.

```
(DEFUN LOOP (NLAMBDA FORM-LST
  (eval-loop FORM-LST) ))

(DEFUN eval-loop (LST RSLT)
  ((ATOM LST)
    (eval-loop FORM-LST) )
  ((OR (ATOM (CAR LST)) (ATOM (CAAR LST))
       (EQ (CAAAR LST) 'LAMBDA))
    (eval-loop (CDR LST) (EVAL (CAR LST))) )
  ((OR (ATOM (CAAAR LST))
       (EQ (CAAAAR LST) 'LAMBDA))
    ((SETQ RSLT (EVAL (CAAR LST)))
      (eval-progn (CDAR LST) RSLT) )
    (eval-loop (CDR LST)) )
  (eval-loop (CDR LST) (eval-progn (CAR LST))) )
```

PROG1 [$form_1$, $form_2$, ..., $form_n$] Special form
PROG2 [$form_1$, $form_2$, ..., $form_n$] Library Macro

(PROG1 $form_1$ $form_2$ ... $form_n$) evaluates $form_1$, evaluates the remaining forms using an implicit PROGN, and returns the result of the evaluation of $form_1$.

(PROG2 $form_1$ $form_2$ ... $form_n$) evaluates $form_1$, evaluates $form_2$, evaluates the remaining forms using an implicit PROGN, and returns the result of the evaluation of $form_2$.

PROG1 and PROG2 sometimes make it possible to avoid introducing a temporary variable to store a result while other expressions are being evaluated.

```
(DEFUN PROG1 (NLAMBDA FORM-LST
  ((NULL FORM-LST) NIL)
  (eval-prog1 (EVAL (CAR FORM-LST))
              (CDR FORM-LST)) ) )

(DEFUN eval-prog1 (RSLT LST)
  (eval-progn LST)
  RSLT )

(DEFMACRO PROG2 (FORM1 . FORMS)
  (LIST 'PROGN FORM1 (CONS 'PROG1 FORMS)) )

$ (SETQ FOO '(A B C D)).                    ;Initialize FOO
(A B C D)

$ (PROG1 (CAR (LAST FOO))
         (SETQ FOO (NBUTLAST FOO)))
D                                ;Remove and return last element

$ FOO                                       ;Last element removed
(A B C)
```

IDENTITY [*object*] Function

(IDENTITY *object*) simply returns *object*. The predicate of an implicit COND must be a function application, not merely a variable. Thus IDENTITY is useful for using a variable as a predicate.

```
(DEFUN IDENTITY (OBJ)
  OBJ )

$ (IDENTITY NIL)
NIL

$ (IDENTITY 'DOG)
DOG

$ (IDENTITY '(A B C))
(A B C)

$ (MAPCAN 'IDENTITY '(A (B) C (D E) F))
(B D E)                                  ;Filter atoms out of list
```

IF [*testform*, *thenform*, *elseform*] — Special form
WHEN [*testform*, $form_1$, $form_2$, ..., $form_n$] — Library Macro
UNLESS [*testform*, $form_1$, $form_2$, ..., $form_n$] — Library Macro

If the value of *testform* is nonNIL, (IF *testform thenform elseform*) evaluates and returns *thenform*. Otherwise, IF evaluates and returns *elseform* or NIL if the *elseform* argument is omitted. The special form IF is the LISP version of the if-then-else construct found in most modern programming languages.

If the value of *testform* is nonNIL, (WHEN *testform* $form_1$ $form_2$ ... $form_n$) evaluates and returns the forms using an implicit PROGN. If *testform* is NIL, WHEN returns NIL without evaluating the forms.

If the value of *testform* is NIL, (UNLESS *testform* $form_1$ $form_2$ ... $form_n$) evaluates and returns the forms using an implicit PROGN. If *testform* is nonNIL, UNLESS returns NIL without evaluating the forms.

```
(DEFMACRO IF (TEST THEN ELSE)
  (LIST 'PROGN (LIST TEST THEN) ELSE) )

(DEFMACRO WHEN (TEST . BODY)
  (LIST 'IF TEST (CONS 'PROGN BODY)) )

(DEFMACRO UNLESS (TEST . BODY)
  (LIST 'IF TEST NIL (CONS 'PROGN BODY)) )
```

Several of the formal definitions in this chapter provide good examples of IF used as a control construct.

COND [$cond_1$, $cond_2$, ..., $cond_n$] Special form

(COND $cond_1$ $cond_2$ ... $cond_n$) successively evaluates the car of each cond form (i.e. a predicate) until either a nonNIL value is encountered or all the predicates have been evaluated. In the former case, COND evaluates the cdr of the cond form with a nonNIL predicate as a list of forms using an implicit PROGN. If the cdr of the nonNIL cond form is empty, COND returns the value of the predicate. If all predicates have been evaluated and all return NIL, COND returns NIL.

In muLISP the implicit COND has greatly diminished the need to explicitly use the COND function. However, as the example shows, COND is useful when making a conditional statement within nested function calls.

```
(DEFUN COND (NLAMBDA COND-LST
  (eval-cond COND-LST) ))

(DEFUN eval-cond (LST RSLT)
  ((ATOM LST) NIL)
  ((SETQ RSLT (EVAL (CAAR LST)))
    (eval-progn (CDAR LST) RSLT) )
  (eval-cond (CDR LST)) )

$ (SETQ NUM -3)
-3

$ (SETQ SIGN (COND
      ((PLUSP NUM) 'POSITIVE)
      ((MINUSP NUM) 'NEGATIVE)
      ((ZEROP NUM) 'ZERO)
      ('NONNUMBER) ) )
NEGATIVE
```

CASE [*keyform*, $case_1$, ..., $case_n$] Library Macro

(CASE *keyform* $case_1$ ... $case_n$) evaluates *keyform* to produce a key. This key is compared with the car of each case clause looking for a match. A match occurs if the car of the case clause is the symbol T or the symbol OTHERWISE, if it is a nonNIL atom and key is EQL to the atom, or if it is a list (i.e. NIL or a cons) and key is EQL to any member of the list. If a match is found, CASE evaluates the cdr of the matching case clause as a list of forms using an implicit PROGN. If the cdr of the matching case clause is empty, CASE returns NIL. If no match is found, CASE returns NIL.

```
(DEFMACRO CASE (KEYFORM . CASES)
  (LIST
    (LIST 'LAMBDA '(KEY) (CONS 'COND
      (MAPCAR '(LAMBDA (CASE) (CONS
          (IF (MEMBER (CAR CASE) '(T OTHERWISE))
              T
              (LIST (IF (ATOM (CAR CASE))
                        'EQL
                        'MEMBER)
                    KEY
                    (LIST 'QUOTE (CAR CASE))) )
          (OR (CDR CASE) '(NIL))) )
        CASES)))
    KEYFORM) )

$ (CASE (+ 2 3)
     (0 'ZERO)
     (1 'ONE)
     ((2 4 6 8) 'EVEN)
     ((3 5 7 9) 'ODD)
     (T '?) )
ODD
```

LET [*letlist*, $form_1$, $form_2$, ..., $form_n$] Library Macro
LET* [*letlist*, $form_1$, $form_2$, ..., $form_n$] Library Macro

If *letlist* is a list of *m* elements of the form (var_i $value_i$), (LET *letlist* $form_1$ $form_2$ ... $form_n$) evaluates the expressions $value_1$ through $value_m$ in order, and temporarily binds the resulting values to the variables var_1 through var_m. Next LET evaluates the forms $form_1$ through $form_n$ using an implicit PROGN, restores the original values to the variables var_1 through var_m, and returns the value of $form_n$.

If *letlist* is a list of *m* elements of the form (var_i $value_i$), (LET* *letlist* $form_1$ $form_2$ ... $form_n$) evaluates the expression $value_1$ and temporarily binds the resulting value to the variable var_1, evaluates the expression $value_2$ and temporarily binds the resulting value to the variable var_2, and so on through var_m. Next LET* evaluates the forms $form_1$ through $form_n$ using an implicit PROGN, restores the original values to the variables var_1 through var_m, and returns the value of $form_n$.

In short, LET and LET* operate identically, except that LET binds the variables in parallel, whereas LET* binds the variables sequentially. See the COMMON.LSP library file for the formal definition of the LET and LET* macros.

```
$ (SETQ X 3)
3

$ (LET ((X 5)
        (Y X))
    (+ X Y) )
8

$ (LET* ((X 5)
         (Y X))
    (+ X Y) )
10

$ X
3
```

DO [*letlist*, $form_1$, $form_2$, ..., $form_n$] Library Macro
DO* [*letlist*, $form_1$, $form_2$, ..., $form_n$] Library Macro

If *letlist* is a list of *m* elements of the form (var_i $value_i$ $step_i$), (DO *letlist* $form_1$ $form_2$ ... $form_n$) evaluates the expressions $value_1$ through $value_m$ in order, and temporarily binds the resulting values to the variables var_1 through var_m. Next DO evaluates the forms $form_1$ through $form_n$ using an implicit PROGN. If all the forms are evaluated without encountering a nonNIL implicit COND predicate, DO evaluates the expressions $step_1$ through $step_m$ in order, and temporarily binds the resulting values to the variables var_1 through var_m. DO continues to evaluate the forms and rebind the variables until a nonNIL implicit COND predicate is encountered. Then DO evaluates the body of the implicit COND, restores the original values to the variables var_1 through var_m, and returns the result of the implicit COND.

DO* operates identically to DO except that DO* binds the variables sequentially rather than in parallel.

Note that in muLISP any or all of the forms in a DO or DO* loop can have conditional exit tests, whereas in Common LISP $form_1$ must be the one and only conditional exit test. See the COMMON.LSP library file for the formal definition of the DO and DO* macros.

```
$ (DO ((LST '(A B C D) (CDR LST))
       (RSLT NIL (CONS (CAR LST) RSLT)))
    ((NULL LST) RSLT) )
(D C B A)                              ;Reverse of list

$ (DO ((X 1 (ADD1 X))
       (Y 0 (+ Y X)))
    ((= X 10) Y) )
45                                     ;Sum of 1 thru 9
```

DOLIST [(*var listform resultform*), $form_1$, ..., $form_n$] Library Ma
DOTIMES [(*var countform resultform*), $form_1$, ..., $form_n$] Library Ma

(DOLIST (*var listform resultform*) $form_1$ ... $form_n$) evaluates *listfo* binds *var* to the first element of *listform*, and evaluates the forms *fo* through $form_n$ using an implicit PROGN. Next DOLIST binds *var* to second element of *listform*, reevaluates the forms, and so on until the is exhausted. Finally, DOLIST binds *var* to NIL, evaluates *resultform* if given, restores *var* to its original value, and returns the value of *resultform* if given or NIL otherwise.

(DOTIMES (*var countform resultform*) $form_1$ ... $form_n$) evalua *countform* and binds *var* to 0. If *var* < *countform*, DOTIMES evalu the forms $form_1$ through $form_n$ using an implicit PROGN. N DOTIMES binds *var* to 1 and if *var* < *countform*, reevaluates the for and so on until *var* ≥ *countform*. Finally, DOTIMES evaluates *resultf* if given, restores *var* to its original value, and returns the value of *resultform* if given or NIL otherwise.

See the COMMON.LSP library file for the formal definition of the DOLIST and DOTIMES macros

```
$ (DOTIMES (X 5)                    ;Display square of 1, 2, 3, 4
          (PRIN1 X) (SPACES 2) (PRINT (* X X))
0  0
1  1
2  4
3  9
4  16
NIL
```

CATCH [*tag*, $form_1$, ..., $form_n$] Special form

After evaluating *tag*, (CATCH *tag* $form_1$ ... $form_n$) evaluates the forms beginning with $form_1$ using an implicit PROGN and returns the result of the last form evaluated, unless an uncaught throw occurs during the evaluation of the forms. If such a throw occurs, and if *tag* is NIL or equal to *THROW-TAG* (i.e. the tag specified by the throw), CATCH deactivates the throw and immediately returns the value specified by the throw. If a throw occurs that does not match *tag*, the throw continues uncaught by this catch.

If the catch tag is NIL, all throws are caught. However, the throw responsible can still be determined by examining the value of the variable *THROW-TAG* (see THROW).

The default muLISP read-eval-print loop catches throws to the tags RETURN and DRIVER. The top level loop catches all throws not otherwise caught (see Section 5.19). See the definition of the function DRIVER for a good example of the use of CATCH.

```
(DEFUN CATCH (NLAMBDA LST
  ((NULL LST) NIL)
  (eval-catch (EVAL (CAR LST)) (CDR LST)) ))

(DEFUN eval-catch (TAG LST RSLT)
  (SETQ RSLT (eval-progn LST))
  ((if no throw is active) RSLT)
  ((OR (NULL TAG) (EQ TAG *THROW-TAG*))
    (deactivate throw)
    (return throw value) )
  (continue throw) )
```

COMMENT [*comments*] Special form

(COMMENT *comments*) ignores its arguments and returns NIL. The words making up the comment must be *not* contain escape or macro characters (see Section 5.13) so they will be accepted by the muLISP reader.

The function COMMENT provides a means of including comments in a function definition. However, such comments use D-code storage space and slow execution speed slightly.

In contrast, comments delimited by comment macro characters are ignored by the muLISP reader when they are read in (see Section 5.13). Although they require disk storage space, such comments do *not* use D-code storage space or slow execution speed.

```
(DEFUN COMMENT (NLAMBDA ()
  NIL ))

$ (DEFUN FOO (X)
    (COMMENT The definition of FOO contains a
        comment that refers to itself!)
    (CONS X X) )
```

UNWIND-PROTECT [$form_1$, $form_2$, ..., $form_n$] Special form

(UNWIND-PROTECT $form_1$ $form_2$ ... $form_n$) evaluates $form_1$ and then evaluates the rest of the forms using an implicit PROGN even if evaluation of $form_1$ was terminated by a throw. When the implicit PROGN is complete, UNWIND-PROTECT returns the result returned by the evaluation of $form_1$ if the evaluation terminated normally. Otherwise, UNWIND-PROTECT continues the throw if the evaluation of $form_1$ was terminated by a throw.

UNWIND-PROTECT is useful for those situations where it is essential to properly terminate or "clean-up" after evaluating $form_1$ no matter what caused it to terminate. For example, the following code could be used to ensure that the precision is restored to its original value following the computation of pi to a high accuracy even if the computation was aborted by a keyboard interrupt or a Memory Full error break.

See the definition of the function DRIVER in Section 5.19 for another example of the use of UNWIND-PROTECT.

```
$ (DEFUN COMPUTE-PI (NEW-PRECISION
      OLD-PRECISION)
    (SETQ OLD-PRECISION (PRECISION NEW-PRECISION))
    (UNWIND-PROTECT (PI)
                    (PRECISION OLD-PRECISION)) )
```

THROW [*tag*, *form*] Special form
RETURN [*form*] Special form
THROW-TAG Variabl

(THROW *tag form*) evaluates *tag* and *form*, sets the variable *THROW TAG* to the value of *tag*, terminates function body evaluation, an unwinds the variable and control stacks back to the most recent CATCI having a catch label eq to the value of *tag* or a catch label of NIL. Th CATCH returns the value of *form*. If not caught by a user program o the read-eval-print loop, control returns to the muLISP top-level loo (see Section 5.19).

(RETURN *form*) evaluates *form*, terminates function body evaluatio and unwinds the variable and control stacks back to the call on th function or LAMBDA body whose definition contains the call o RETURN. This call returns the value of *form*. Since there is an implici catch of RETURN around all function calls, RETURN is implemented b making a throw tagged RETURN.

Note that although THROW and RETURN are special forms, thei arguments are evaluated *before* the throw is made. Besides erro handling, CATCH and THROW are useful for making it possible to exit nest of control constructs without actually leaving the function RETURN does.

See the definition of the function BREAK in Section 5.19 for a le contrived example of the use of THROW.

```
(DEFUN THROW (TAG FORM)
  (SETQ *THROW-TAG* TAG)
  (unwind the variable and control stacks
      until caught by a matching CATCH that
      returns FORM) )

(DEFUN RETURN (FORM)
  (THROW 'RETURN FORM) )
```

RESTART [] Function
SYSTEM [*n*] Function

(RESTART) closes all open files, abandons the current muLISP environment, and initiates a fresh muLISP system. All variable bindings, user-defined functions, and property values in the current environment are destroyed.

(SYSTEM) closes all open files, terminates execution of muLISP, and returns control to the host operating system.

If *n* is zero or a positive integer less than 256, (SYSTEM *n*) returns an exit code (also called a return code) of *n* to its host process.

The exit code can be interrogated by the MS-DOS batch command "IF ERRORLEVEL" (see the Batch Processing Commands section of the MS-DOS Reference Manual for details). If *n* is omitted, muLISP returns the default exit code stored in the muLISP Base Page (see the Appendix to this manual).

```
(DEFUN SYSTEM (BYTE)
  (close all currently open files)
  ((AND (INTEGERP BYTE) (<= 0 BYTE 255))
    (exit to host returning the exit code BYTE) )
  (exit to the host returning default exit code) )

$ (RESTART)              ;Restart muLISP with a virgin environment

$ (SYSTEM)                  ;Quit muLISP, exit to operating system

A>                                                  ;The DOS prompt
```

EXECUTE [*program, command*] — Function
DOS [*command*] — Library Function

(EXECUTE *program command*) compacts and suspends muLISP, then loads and executes *program* passing it *command* as an argument. When *program* terminates, control reverts to muLISP and EXECUTE returns *program*'s exit code. If *program* is not found, EXECUTE returns NIL.

(DOS *command*) invokes the MS-DOS command processor (see the MS-DOS Reference Manual) to process *command* just as if *command* were entered following the MS-DOS prompt. (DOS) suspends muLISP and invokes the MS-DOS command processor which displays the MS-DOS prompt. MS-DOS commands can be entered until an EXIT command returns control to muLISP. DOS is defined in COMMON.LSP.

```
(DEFUN DOS (COMMAND)
  (EXECUTE (GETSET 'COMSPEC)
           (IF COMMAND (PACK* "/C " COMMAND))) )

$ (DOS "DIR B:/W")                    ;Display drive B's directory
<B's directory displayed>
0                                     ;DOS returns exit code

$ (DOS)                               ;Suspend muLISP, start-up MS-DOS
MS-DOS Version 2.10                   ;Displays MS-DOS prompt

A>EXIT                                ;Issue MS-DOS EXIT command
0                                     ;DOS resumes execution
```

Using EXECUTE and DOS, the full range of facilities provided by MS-DOS and application programs can be accessed, all from within muLISP without abandoning the current muLISP environment. Communication with the "child" process can be accomplished through the use of source files, SYS files, and/or exit codes. For example, the following command starts running another rendition of muLISP and "passes" it the file EXPERT:

```
$ (EXECUTE "B:MULISP.COM" "EXPERT")
```

5.18 MEMORY MANAGEMENT FUNCTIONS

The function RECLAIM forces a garbage collection and returns the resulting amount of free data space. Since memory management is fully automatic in muLISP, RECLAIM is normally used only to determine the amount of free space, or prior to a timing test to minimize the number of garbage collections during the test. The muLISP memory management system is described in Chapter 4.

RECLAIM [] Function

(RECLAIM) forces a garbage collection to reclaim the inactive data structures in the four muLISP data spaces. If required, a memory reallocation is also performed. If muLISP is running on an IBM PC, the blinking cursor temporarily changes to a full block during the collection and reallocation. RECLAIM returns the total number of bytes available in the atom, vector, pointer, and stack spaces.

```
(DEFUN RECLAIM ()
  (IF *GCSTAT*
      (display pre-GC statistics) )
  (perform garbage collection on all data spaces)
  (IF *GCSTAT*
      (display post-GC statistics) )
  (IF (available free space is not optimally pro-
        portioned between the four data spaces)
      (PROGN
        (reallocate free memory among the spaces)
        ((NOT *GCSTAT*))
        (display post-allocation statistics) ) )
  (return the amount of free space in bytes) )
```

```
(RECLAIM)                          ;Force a garbage collection
340679                             ;Return amount of free space
```

GCSTAT: NIL Control varia

If *GCSTAT* is nonNIL, informative GC statistics are displayed on t console before and after all garbage collections and memory reallocatio If *GCSTAT* is NIL, no GC statistics are displayed.

The first line of the GC statistics shows the available space just bef the garbage collection. The second line shows the available space af the collection. Lines beginning with "GC" indicate garbage collecti statistics; lines beginning with "RA" indicate data space reallocati statistics. The format for a line of GC statistics is as follows:

GC: NNNN AAAA/AAAA VVVV/VVVV PPPP/PPPP SSSS/SSSS TTTT/TTTT

NNNN, AAAA, VVVV, PPPP, SSSS, and TTTT denote hexadecimal numbe NNNN is the number of garbage collections since muLISP began executi The first AAAA, VVVV, PPPP, and SSSS of each pair of numbers is t amount of free space in bytes in the atom, vector, pointer, and sta spaces respectively. Chapter 4 describes the purpose of each of the spaces. The first TTTT of the pair is the total amount of free space (i the sum of AAAA, VVVV, PPPP, and SSSS). The numbers following the " are the total size of the respective spaces, including both the free a used space.

Two bytes are required for each muLISP pointer cell in the atom, point and stack spaces. One byte is required for each character in a print na and two bytes are required to store the length of a print name. Numl vectors use only as many words as necessary to store the number in binary.

The GC statistics are heavily dependent on the implementation of muL being run and should normally be of no concern to users. However, th are sometimes useful for determining what is causing the system to thra (i.e. to conduct an excessive number of garbage collections for very lit free space returned). GC statistics are also displayed when a Mem Full error occurs to show which space was responsible (see Chapter 4).

```
$ (SETQ *GCSTAT* T)                    ;Turn on GC statistics flag
T

$ (LOOP (OBLIST))                      ;Create lots of conses

GC:  0001  A7C0/B4C0  1ECCB/20000  0D120/274C0  2ABC/2AE0  39065/55460
GC:  0001  A7C8/B4C0  1ECCD/20000  26D20/274C0  2ABC/2AE0  52C6F/55460
RA:  0001   0208/0F00  1ECCD/20000  2C180/2C920  283C/2860  4D88F/50080

GC:  0002   0208/0F00  1ECCD/20000  0D3A0/2C920  283C/2860  2EAAF/50080
GC:  0002   0208/0F00  1ECCD/20000  2C21C/2C920  283C/2860  4D92B/50080

GC:  0003   0208/0F00  1ECCD/20000  0D3A0/2C920  283C/2860  2EAAF/50080
GC:  0003   0208/0F00  1ECCD/20000  2C21C/2C920  283C/2860  4D92B/50080
```

This "endless" loop and the display of GC statistics will continue until a console interrupt is initiated by pressing the ESC key.

The three lines of GC statistics for garbage collection number 1 indicate that a memory reallocation occurred. More memory was allocated to the pointer space because of the heavy demand by OBLIST for new conses. Subsequent garbage collections did not perform reallocations because the data space sizes were already optimized. However, if the program changed and started using the atom space, for example, another reallocation would occur to expand the size of that region at the expense of the others.

The example shown above was done on an 80286 based computer with 640 Kilobytes of memory. The numbers will vary greatly depending on the amount of memory available to muLISP.

GCHOOK: NIL Control varia

If *GCHOOK* is the name of a function, after each garbage collecti the function is called. If *GCHOOK* is NIL, no call is mad Otherwise, an Undefined Function error break occurs.

By assigning to *GCHOOK* a function that monitors the amount of fr space after garbage collections, programs may be able to automatical prevent thrashing and/or Memory Full errors by the release of unneede data when the amount of free space becomes dangerously low.

FREE-LIST: NIL Variabl

The constructor functions are required to create new conses. If *FREE LIST* is an atom, space for the new conses is taken from the area o memory called the pointer space. If previous consing has exhausted th pointer space, a garbage collection is automatically performed to reclai the space used by conses that are no longer accessible.

However, if *FREE-LIST* is a cons, that cons is used when a new con is required by muLISP. By nconcing no longer needed lists to *FREE LIST*, application programs can reduce or eliminate the occurrence o garbage collections.

Great care must be taken to insure that conses nconced to *FREE-LIST are not going to be referenced again. We recommend that *FREE-LIST not be incorporated into an application program until the program i otherwise thoroughly debugged.

The following example adds 6 (*not* 9) conses to *FREE-LIST*:

```
$ (SETQ JUNK '(A B (C D E) 3 4 5))
(A B (C D E) 3 4 5)

$ (SETQ *FREE-LIST* (NCONC JUNK *FREE-LIST*))
(A B (C D E) 3 4 5)
```

ALLOCATE [*n*] Function

If *n* is a positive integer, (ALLOCATE *n*) frees *n* bytes of memory in the 8086 code segment following the muLISP machine code and returns the base offset address of the newly allocated space. If there is insufficient room to free *n* bytes, ALLOCATE returns NIL.

(ALLOCATE) returns the current base offset address of the beginning of free space in the 8086 code segment following the muLISP machine code.

ALLOCATE is primarily intended for compilers that generate machine code.

```
(DEFUN ALLOCATE (N)
  ((PLUSP N)
  ((there is sufficient free memory)
    (free N bytes of memory after machine code)
    (return base offset address of free space) ) )
  (return base offset address of free space) )

$ (ALLOCATE 100)                    ;Free 100 bytes of memory
39120                               ;Free memory begins at offset 39120

$ (ALLOCATE)                        ;Return start of next free space
39232
```

NEW-CODE-SPACE [] Functio

(NEW-CODE-SPACE) creates a new D-code space for the storage o function definitions and returns T. There can be as many as six D-cod spaces. If six D-code spaces already exist or there is insufficient memor for a new space, NEW-CODE-SPACE returns NIL.

Provided there is sufficient computer memory, each D-code space ca store about 55K of D-code. This allows muLISP to take advantage of th entire 640K MS-DOS work space and makes it possible to load well ove 25000 lines (about 500 pages) of LISP source code. The judicious use o NEW-CODE-SPACE can also avoid "Memory Full" errors (see Section 4.5 that are caused by the exhaustion of the Atom Space (see Section 4.2).

muLISP automatically creates a new space when function definitions hav filled the current D-code space. If a function in one D-code space calls function in another space, a slight speed penalty must be paid whil muLISP switches to the new space. To reduce the effect of this D-cod space "fault", large programs (i.e. those which cannot be stored in a singl D-code space) can be divided into modules that operate relativel independently of one another so that each module can be loaded into separate D-code space.

Thus to make maximum use of available memory, simply break up th source file for large programs into modules each less than 80 pages i length. Then load the first module, call NEW-CODE-SPACE, load th second module, call NEW-CODE-SPACE again, etc.

If only a single D-code space is being used and full condensing has no been in effect, the space used to store the original definition of function being redefined is automatically reclaimed (see *CONDENSE* i Section 5.10). However, if additional D-code spaces have been create (either automatically or by NEW-CODE-SPACE) and a function i redefined, the space for the original definition is *not* reclaimed.

```
$ (NEW-CODE-SPACE)                ;Create a new D-code spac
T
```

5.19 USER INTERFACE FUNCTIONS

The functions in this section provide the basic user interface to muLISP. After initializing its data space, muLISP enters the top-level loop. This loop sets an internal break-level counter to 0 and then calls the current driver function by issuing the call (FUNCALL 'DRIVER *command-line*) where *command-line* is the MS-DOS command line that was used to initiate muLISP. The command line can be used to pass a file name to a program's driver function. For instance, the muLISP editor takes advantage of this feature.

Upon return from the driver function, the top-level loop catches all throws not previously caught by the application program or the current driver function, and deactivates the throw. If *THROW-TAG* is the symbol SYSTEM, the top-level loop closes all open files, restores the normal cursor shape (IBM PCs only), terminates muLISP, and returns control to MS-DOS. If *THROW-TAG* is the symbol "Disk Full" or "Memory Full", the top-level loop generates an appropriate error break.

```
(LOOP
  (set the break-level counter to 0)
  (CATCH NIL (FUNCALL DRIVER))
  ( ((throw was not active?))
    ((EQ *THROW-TAG* 'SYSTEM)
      (close all currently open files)
      (restore the normal cursor shape)
      (terminate muLISP) )
    ((EQ *THROW-TAG* "Memory Full")
      (display GC statistics)
      (BREAK NIL "Memory Full") )
    ((EQ *THROW-TAG* "Disk Full")
      (BREAK NIL "Disk Full) ) ) )
```

DRIVER [*filename*]	Functi
+	Variabl
++	Variabl
+++	Variabl
*	Variabl
**	Variabl
***	Variabl
-	Variabl

DRIVER implements the standard muLISP read-eval-print loop. It i tl default driver function called by the top-level loop. DRIVER is al called when an error or interrupt break occurs and the user selects tl Break option (see the function BREAK).

If *filename* is nonNIL and names an existing SYS or source fil (DRIVER *filename*) attempts to load *filename* (see LOAD in Sectio 5.15). If *filename* is nonNIL but no such file exists, DRIVER displa the "File not found" error message (see Section 4.5). If *filename* is NII DRIVER goes straight to the following read-eval-print loop:

If internal break-level counter is non-zero (i.e. if DRIVER was not calle directly by the top-level loop), DRIVER displays the counter. Ne:t displays the muLISP prompt (a dollar sign ($) character followed by space). Finally it reads an expression from the console using the functio READ, evaluates the expression using the function EVAL, and sends tl result to the console using the function PRINT.

As a programming convenience, DRIVER assigns the last thr expressions it reads to the variables +, ++, and +++ respectively. Al o assigns the result of the last three expressions it evaluates to the variabl *, **, and *** respectively. Finally it assigns the expression it has re most recently to the variable - before it evaluates the expression. Tl example on the next page illustrates the use of these variables.

DRIVER catches throws to the tag DRIVER that may occur during tl reading, evaluating, or printing of expressions. This feature makes possible for functions such as BREAK to abort program execution an immediately pass control to the current level read-eval-print loop.

DRIVER repeats the read-eval-print loop until the user enters the expression (RETURN *form*) or presses the ESC key. After either of these actions, DRIVER terminates the loop, and returns the result of evaluating *form* in the former case or returns NIL in the latter case.

```
(DEFUN DRIVER (FILE-NAME)
  ( ((NULL FILE-NAME))                          ;File name given?
    ((LOAD FILE-NAME))                                ;Load file
    (WRITE-LINE "File not found" T) )                 ;No file
  (LOOP
    (TERPRI NIL T)
    ( ((ZEROP break-level))
      (WRITE break-level T) )
    (WRITE-STRING "$ " T)
    (CATCH NIL
      (SETQ - (READ T))
      (PSETQ * (UNWIND-PROTECT
                 (EVAL -)
                 (SETQ +++ ++
                       ++  +
                       +   -) )
               *** **
               **  *)
      (WRITE * T)
      (TERPRI NIL T) )
    ( ((throw was not active?))
      ((EQ *THROW-TAG* 'DRIVER)
        (deactivate throw) )
      ((EQ *THROW-TAG* 'RETURN)
        (deactivate throw)  (RETURN *) )
      (THROW *THROW-TAG* *) ) ) )

$ (CONS 'SAM '(ANN JOE SUE))                      ;Sample interaction
(SAM ANN JOE SUE)

$ +                                               ;Get previous input
(CONS (QUOTE SAM) (QUOTE (ANN JOE SUE)))

$ **                                              ;Get previous output
(SAM ANN JOE SUE)
```

DRIVER: 'DRIVER — Control variable

The initial value of the *control variable* driver is the *function* DRIVER that implements the read-eval-print loop described on the preceding pages. However, the default driver function can be replaced with a new driver function by setting the *variable* DRIVER to the name of the new function. Then the command (RETURN) will terminate the default driver function and the top-level loop will automatically call the new driver function.

This example shows how to implement an eval-quote LISP read-eval print loop:

```
$ (DEFUN EVAL-QUOTE ()                    ;Define eval-quote loop
    (CATCH 'RETURN (LOOP
      (CATCH 'DRIVER
        (TERPRI NIL T)
        (WRITE-STRING "> " T)
        (PRINT (APPLY (READ) (READ)))) ))) )
EVAL-QUOTE

$ (SETQ DRIVER 'EVAL-QUOTE)               ;Reassign DRIVER
EVAL-QUOTE

$ (RETURN)                                ;Exit from default loop

> CONS (DOG (CAT COW PIG))                ;Eval-quote interaction
(DOG CAT COW PIG)
```

BREAK [*object*, *message*] Function
BREAK Variable

(BREAK *object message*) suspends program execution and displays the symbol *message*, the string " Break: ", and *object* on the console screen to indicate the cause of the break. If *message* is NIL or omitted, this line is not displayed. On the next line the break option prompt

```
Continue, Break, Abort, Top-level, Restart, System
```

is displayed. The system then waits for the user to select one of the options by entering the first letter of its name (C, B, A, T, R, or S). Note that the options are listed in order of increasingly drastic action. If after a few seconds the user fails to select an option, a short beep tone is generated to inform the user of the occurrence of a break.

Continue: *object* is returned as the value of the break and execution continues as before.

Break: temporarily suspends execution for a break. The variable BREAK is set to *object*, and the internal break-level counter is incremented. The function DRIVER is called to permit the user to examine the environment present at the time the break occurred. Note that DRIVER will display the nonzero break-level counter in the driver prompt as a reminder that a break is in progress.

When ready to exit the break and return to the calling program, issue the command (RETURN *form*). It evaluates *form*, exits the break, and returns the result of the evaluation as the value of the break. For example, the following command evaluates and returns *object*:

```
(RETURN (EVAL BREAK))
```

Just pressing the ESC key also exits the break and returns NIL to the calling program.

Upon return from the break, BREAK decrements the break-level counter, restores the variable BREAK to its original value, and

returns the value of the call to DRIVER as the value of the call to BREAK.

Abort: aborts the evaluation in progress at the time of the break and throws control to the most recent catch labeled DRIVER (usually the read-eval-print loop).

Top-level: aborts the evaluation in progress at the time of the break and throws control to the most recent catch labeled NIL (usually the top-level loop).

Restart: closes all open files, abandons the current muLISP environment, and initiates a fresh muLISP system. This is done only after the user confirms the action by answering Yes to the prompt "Are you sure (Y/N)?".

System: closes all open files, terminates execution of muLISP, and returns control to the operating system. This is done only after the user confirms the action by answering Yes to the prompt "Are you sure (Y/N)?".

When an error condition is detected by muLISP, BREAK is called with the errant function form and an appropriate error message letting the user decide how to proceed. Naturally, BREAK can be redefined to suit the application.

Break-points are calls to the function BREAK in function definitions. When a break-point is encountered in the execution of a program, variable values and function definitions can be examined and/or changed before proceeding with the execution. This can greatly facilitate the debugging of programs. Using the muLISP debugger, function entry break-points can be easily inserted and removed. Using the muLISP editor, break-points can be inserted at key points anywhere within function definitions.

```
(DEFUN BREAK (BREAK MSG
    CHAR )
  ( ((NULL MSG))
    (TERPRI NIL T)
    (PRINC MSG T)
    (WRITE-STRING " Break: " T)
    (WRITE BREAK T)
    (TERPRI NIL T) )
  (WRITE-STRING "Abort, Break, Continue, " T)
  (WRITE-STRING "Top-level, Restart, System? " T)
  (LOOP
    (WRITE-BYTE 7 NIL T)
    (SETQ CHAR (CHAR-UPCASE (ASCII(READ-BYTE T))))
    ((EQ CHAR 'A)
      (THROW 'DRIVER BREAK) )
    ((EQ CHAR 'B)
      (INCQ break-level)
      (CLEAR-INPUT)
      (SETQ BREAK (DRIVER))
      (DECQ break-level)
      BREAK )
    ((EQ CHAR 'C)
      BREAK )
    ((EQ CHAR 'T)
      (THROW NIL BREAK) )
    ((EQ CHAR 'R)
      (RESTART) )
    ((EQ CHAR 'S)
      (SYSTEM) ) ) )
```

Several of the formal definitions in this chapter provide good examples of the use of BREAK.

Y-OR-N-P [*message*] Library Functi
YES-OR-NO-P [*message*] Library Functi

If *message* is a nonNIL symbol, (Y-OR-N-P *message*) displays *messa* on a fresh line immediately followed by a "(Y/N)" prompt, clears t console's input buffer, and waits for the user to enter a "Y" or " (ignoring case). If the user enters a "Y", Y-OR-N-P returns T. If t user enters an "N", it returns NIL.

If *message* is a nonNIL symbol, (YES-OR-NO-P *message*) displa *message* on a fresh line immediately followed by a "(Yes or No)" prom clears the console's line edit buffer, and waits for the user to type "Yes" or "No" (ignoring case) and press RETURN. If the user types "Ye YES-OR-NO-P returns T. If the user types "No", it returns NIL.

These predicates provide a standard convention for asking the user "yes no" questions. Y-OR-N-P should be used for non-dangerous situatio where a simple "Y" or "N" is a sufficient answer. YES-OR-NO-P shou be used for potentially dangerous situations where the user should st and think before answering.

```
(DEFUN Y-OR-N-P (MSG
    CHAR )
  ( ((NULL MSG))
    (FRESH-LINE T)
    (WRITE-STRING (PACK* MSG " (Y/N) ") T) )
  (CLEAR-INPUT T)
  (LOOP
    (SETQ CHAR (CHAR-UPCASE (ASCII(READ-BYTE T))))
    ((EQ CHAR 'Y) (WRITE-STRING CHAR T) T)
    ((EQ CHAR 'N) (WRITE-STRING CHAR T) NIL)
    (WRITE-BYTE 7 NIL T) ) )

$ (Y-OR-N-P "Continue lesson?")
Continue lesson? (Y/N) Y
T

$ (YES-OR-NO-P "Delete your life's work?")
Delete your life's work? (Yes or No) No
NIL
```

LINE-EDIT [*text*, *point*, *column*, *width*]	Function
LINE-TERMINATOR	Variable
LINE-POINT	Variable
LINE-COLUMN	Variable

When reading a line of text from the console, muLISP provides the user with a powerful line editor with many control character commands (see Section 2.3). This line editor is called automatically by the input functions (see Section 5.13). However, the line editor can also be called directly enabling a program to insert text into the line edit buffer and pre-position the cursor within the line of text. This makes it easy to present the user with default entries and to show the user the location of syntax errors.

If *text* is an atom, *point* is zero or a positive integer less than or equal to the length of *text*, *width* is a positive integer less than or equal to the current window width minus the current cursor column position (see Section 5.21), and *column* is zero or a positive integer less than or equal to both *point* and *width*, (LINE-EDIT *text point column width*) displays *text* beginning from the current cursor position for a maximum of *width* columns and positions the cursor at an offset of *column* columns from the beginning of the window. The portion of text displayed is determined such that the cursor is positioned on the *point*'th character in the text.

The default value of *text* is the empty string. The default value of *point* and *column* is zero. The default value of *width* is the current window width minus the current cursor column position.

After the user edits the line of text and presses RETURN or any unrecognized control key or IBM extended function key (see Section 1.5), LINE-EDIT sets *LINE-TERMINATOR* to the byte value of the key that terminated the line edit, sets *LINE-POINT* to the final offset in the line of text, sets *LINE-COLUMN* to the final cursor column position, and returns the line of text as a symbol.

Note that if *text* is the line of text returned by the last call to LINE-EDIT and *point* and *column* are the value of *LINE-POINT* and *LINE-COLUMN* respectively, LINE-EDIT resumes editing as if it had never been exited. Also, programs can invoke functions based on the byte value of the key that terminated the line edit. Based on these facts, it is possible for programs to add new control character commands to those

already provided by the line editor. It is even possible to redefine th primitive commands by modifying the Line Editor Control Key Table an the Line Editor Extended Function Key Table in the mul.ISP base pag (see Appendix B).

INSERT-MODE: T Control variabl

While line editing, if *INSERT-MODE* is nonNIL, new characters ar added to the text rather than replacing old text. If *INSERT-MODE* i NIL, new characters replace old text. Thus, the line editor's CTRL-^ command simply sets *INSERT-MODE* to (NOT *INSERT-MODE*).

Given a default line of text, a column width, and an initial value fo *INSERT-MODE*, the example function GET-LINE presents the use with the text in a one line window of the specified width. The user ca edit the text using all the facilities provided by the mul.ISP line edito (see Section 2.3). If the user presses the RETURN key, the editing i terminated and GET-LINE returns the newly edited line. If the use presses the ESC key, the editing is aborted and GET-LINE returns NIL.

If the user presses any non-recognized control or function key, GET LINE sounds a warning beep before resuming the line-editing. Note tha by testing for other values of *LINE-TERMINATOR*, GET-LINE ca be extended to recognize additional function and/or control key appropriate for a given application.

```
$ (DEFUN GET-LINE (LINE COLUMNS *INSERT-MODE*
     *LINE-POINT* *LINE-COLUMN*)
   (LOOP                                        ;Edit lin
     (SETQ LINE (LINE-EDIT LINE *LINE-POINT*
                  *LINE-COLUMN* COLUMNS))
     ((EQ *LINE-TERMINATOR* 13)                 ;Check for RETUR
       LINE )
     ((EQ *LINE-TERMINATOR* 27)                 ;Check for ES
       NIL )
     (WRITE-BYTE 7 NIL T) ) )                   ;Sound bee
```

5.20 HARDWARE INTERFACE FUNCTIONS

The functions in this section facilitate communication between muLISP programs and the host computer's hardware and operating system. The computer's technical reference and operating system manuals contain the information required to make full use of these muLISP functions.

MEMORY [*address*, *value*, *flag*] Function

If *address* is zero or a positive integer less than the address space of the microprocessor running muLISP, (MEMORY *address*) returns the byte (i.e. the 8 bit value) at *address*.

If *value* is zero or a positive integer less than 256, (MEMORY *address value*) stores the byte *value* at *address* and returns the original byte value.

If *flag* is nonNIL, (MEMORY *address* NIL *flag*) returns the word (i.e. the 16 bit value) at *address*. If *value* is zero or a positive integer less than 65536, (MEMORY *address value flag*) stores the word *value* at *address* and returns the original word value.

> WARNING: MEMORY should be used with care since there is no protection against inadvertently destroying the muLISP interpreter or the host operating system.

```
(DEFUN MEMORY (ADDR VALUE FLAG)
  ((<= 0 ADDR (the address space of processor))
    ((NULL FLAG)
      ((<= 0 VALUE 255)
        (replace the byte at ADDR with VALUE)
        (return the original byte at ADDR) )
      (return the byte at ADDR) )
    ((<= 0 VALUE 65535)
      (replace the word at ADDR with VALUE)
      (return the original word at ADDR) )
    (return the word at ADDR) ) )
```

```
$ (SETQ *PRINT-BASE* 16 *READ-BASE* 16)
10                                          ;Set hexadecimal I/O

$ (MEMORY 0A)                               ;Get byte at location 0A11
0

$ (MEMORY 0A NIL T)                         ;Get word at location 0A11
0F000

$ (MEMORY 0A 5)                             ;Store a byte at location 0A11
0

$ (MEMORY 0A)                               ;Get byte at location 0A11
5

$ (MEMORY 0A NIL T)                         ;Get word at location 0A11
0F005

$ (MEMORY 0A 0)                             ;Restore original byte at 0A11
5

$ (SETQ *PRINT-BASE* 0A *READ-BASE* 0A)
10                                          ;Set decimal I/O
```

CSMEMORY [*offset*, *value*, *flag*] Function
DSMEMORY [*offset*, *value*, *flag*] Function

CSMEMORY and DSMEMORY are only available if muLISP is running on a computer based on one of the Intel 8086 family of microprocessors. Their function is similar to that of MEMORY, except the first argument is a 16 bit offset into a segment rather than a 20 bit physical address. The CSMEMORY offset is relative to the code segment. The DSMEMORY offset is relative to the data segment.

If *offset* is zero or a positive integer less than 65536, (CSMEMORY *offset*) and (DSMEMORY *offset*) return the byte at *offset*. If *value* is zero or a positive integer less than 256, (CSMEMORY *offset value*) and (DSMEMORY *offset value*) store the byte *value* at *offset* and returns the original byte value.

If *flag* is nonNIL, (CSMEMORY *offset* NIL *flag*) and (DSMEMORY *offset* NIL *flag*) return the word at *offset*. If *value* is zero or a positive integer less than 65536, (CSMEMORY *offset value flag*) and (DSMEMORY *offset value flag*) store the word *value* at *offset* and returns the original word value.

(CSMEMORY) and (DSMEMORY) return the contents of the current muLISP code segment and data segment registers respectively.

> WARNING: These functions should be used with care since they can destroy the muLISP system.

```
(DEFUN CSMEMORY (OFFSET VALUE FLAG)
  ((<= 0 OFFSET 65535)
   ((NULL FLAG)
     ((<= 0 VALUE 255)
       (replace the byte at OFFSET with VALUE)
       (return the original byte at OFFSET) )
     (return the byte at OFFSET) )
    ((<= 0 VALUE 65535)
      (replace the word at OFFSET with VALUE)
      (return the original word at OFFSET) )
    (return the word at OFFSET) )
  (return the current CS register) )
```

CSMEMORY is useful for accessing and/or changing variables stored i the muLISP base page (see Appendix B). The multiple escape charac te that muLISP uses to delimit print name strings containing specia characters is stored in the base page at offset 849. This example show how to change the multiple escape character.

```
$ (ASCII (CSMEMORY 849))
\|                                    ;Default multiple escape cha

$ "LISP is fun."
|LISP is fun.|                        ;String delimited by | cha

$ (CSMEMORY 849 (ASCII \"))           ;Change multiple escape cha
124                                   ;Return original cha

$ "LISP is fun."
"LISP is fun."                        ;String delimited by " cha

$ (CSMEMORY 849 124)
34                                    ;Restore original multiple escape cha
```

DSMEMORY is useful for accessing variables stored in the muLISP global storage area (see Appendix C). Warning: The value of variables i the global storage area should *not* be changed. The beginning and endin addresses of the D-code is stored as words in the global storage area a offsets 34 and 36 respectively. This example shows how to determine th current size of the D-code in bytes (naturally the results will vary).

```
$ (DSMEMORY 34 NIL T)                 ;Base of D-cod
5346

$ (DSMEMORY 36 NIL T)                 ;End of D-cod
24534

$ (- * **)                            ;Bytes of D-cod
19188
```

PORTIO [*port*, *value*, *flag*] Function

If *port* is zero or a positive integer less than 65536, (PORTIO *port*) inputs and returns a byte value from *port*. If *value* is zero or a positive integer less than 256, (PORTIO *port value*) outputs *value* to *port* and returns NIL.

If *flag* is nonNIL, (PORTIO *port* NIL *flag*) inputs and returns a word value from *port*. If *value* is zero or a positive integer less than 65536, (PORTIO *port value*) outputs *value* to *port* and returns NIL.

This function is used to send byte or word values to the output ports, and to receive byte or word values from input ports. This can be useful for muLISP programs that need to communicate with external devices such as sensors, robots, etc.

```
(DEFUN PORTIO (PORT VALUE FLAG)
  ((<= 0 PORT 65535)
    ((NULL FLAG)
      ((<= 0 VALUE 255)
        (output byte VALUE to PORT)
        NIL )
      (input a byte value from PORT) )
    ((<= 0 VALUE 65535)
      (output word VALUE to PORT)
      NIL )
    (input a word value from PORT) ) )
```

```
$ (PORTIO 70)                          ;Input byte from port 70
208                                          ;(results will vary)

$ (PORTIO 70 NIL T)                    ;Input word from port 70
36610                                        ;(results will vary)

$ (PORTIO 70 10)                        ;Output byte to port 70
NIL

$ (PORTIO 70 10 T)                      ;Output word to port 70
NIL
```

LOCATION [*object*] **Function**

The pointer cells making up an object are stored in separate data segments at the same offset address. (LOCATION *object*) returns the offset of *object*'s cells from the base of the data segments. Since all cells require exactly two bytes, LOCATION always returns an even integer.

Because muLISP uses address typing to determine the type of an object, the location of a symbol will be less than the location of a number which will be less than the location of a cons. Note that muLISP automatically performs compacting garbage collections and memory reallocations that can change the location of an object. LOCATION is primarily intended for compilers that generate machine code which must refer to the location of symbols in memory.

```
(DEFUN LOCATION (OBJECT)
  (the offset address of OBJECT) )
```

```
$ (LOCATION NIL)                    ;Location of type symbol NIL
256                                 ;(Note: NIL is fixed at 10011)

$ (LOCATION 2)                      ;Location of a number
11622

$ (LOCATION '(A . B))               ;Location of a cons
37838
```

REGISTER [*n*, *m*] Function

(REGISTER *n m*) stores *m* in muLISP pseudo-register *n*, if *m* is a nonnegative integer less than 65536, and returns NIL. If *m* is not such an integer, the content of pseudo-register *n* is returned. The initial value for all the pseudo-registers is zero.

```
(DEFUN REGISTER (N M)
  ((AND (INTEGERP N) (<= 0 N 9))
    ((AND (INTEGERP M) (<= 0 M 65535))
      (store M in register N)
      NIL )
    (the integer value of register N) ) )
```

```
$ (REGISTER 2)                          ;Examine register 2's contents
0

$ (REGISTER 2 100)                      ;Set register 2 to 100
NIL

$ (REGISTER 2)                          ;Examine register 2's contents
100
```

The definition of GET-DATE on the next page provides a good example of the use of REGISTER.

INTERRUPT [*n*] Function

(INTERRUPT *n*) loads the microprocessor's registers from the muLISP pseudo-registers, executes interrupt number *n*, stores the register values returned by the interrupt in the pseudo registers, and returns NIL. Note: INTERRUPT is only available if muLISP is running on a computer based on one of the Intel 8086 family of microprocessors.

There are 10 muLISP pseudo-registers numbered 0 through 9. They correspond to the 8086 registers as follows:

Pseudo:	0	1	2	3	4	5	6	7	8	9
8086:	AX	BX	CX	DX	SI	DI	BP	DS	ES	FLAG

The functions REGISTER and INTERRUPT enable muLISP programs to communicate with external machine language routines via the 256 interrupt vectors provided by the 8086 microprocessor. The computer's technical reference and operating system manuals will describe the services provided by the interrupt vectors.

For example, Interrupt 33 (i.e. 21H) is used to make MS-DOS system calls. If register AH is 42 (i.e. 2AH), Interrupt 33 will return the year in CX (1980-2099), the month in DH (1-12), and the day in DL (1-31). Using this system call, GET-DATE returns the current year, month, and day as a list of three integers (see also GET-DECODED-TIME):

```
$ (DEFUN GET-DATE ()
    (REGISTER 0 (* 42 256))                    ;Set AH to 42
    (INTERRUPT 33)                             ;Make MS-DOS call
    (LIST (REGISTER 2)                         ;CX has year
          (SHIFT (REGISTER 3) -8)              ;DH has month
          (LOGAND (REGISTER 3) 255)) )         ;DL has day

$ (GET-DATE)                                   ;The 1987 New Year
(1987 1 1)
```

BINARY-LOAD [*filename*, *offset*] Function

If *offset* is a positive integer less than 65536 and the file on the drive and directory specified by *filename* exists, (BINARY-LOAD *filename offset*) loads *filename* into the muLISP code segment at *offset* and returns the number of bytes loaded. If no file name extension is specified, the extension BIN is assumed. If *offset* is out of range or no file specified by *filename* exists, BINARY-LOAD returns NIL.

BINARY-LOAD is normally used in conjunction with the function ALLOCATE to load externally assembled or compiled machine language routines into the muLISP code segment memory. Chapter 9 gives an example of its use.

```
(DEFUN BINARY-LOAD (FILE-NAME OFFSET)
  ((AND (INTEGERP OFFSET) (<= 0 OFFSET 65535))
    ((PROBE-FILE FILE-NAME)
      (load a binary image of FILE-NAME into the
          code segment beginning at OFFSET)
      (return bytes loaded) ) ) )
```

SNAPSHOT [*address*, *atom*] Function

If *address* is zero or a positive integer less than the address space of the microprocessor running muLISP and if *atom* is a positive integer less than 65536, (SNAPSHOT *address atom*) returns the symbol whose print name consists of the *atom* bytes of memory beginning at *address*. If *atom* is a symbol, (SNAPSHOT *address atom*) copies the bytes in *atom*'s print name to memory beginning at *address* and returns T.

SNAPSHOT is especially useful for saving and later restoring memory-mapped video screen memory. The IBM PC Monochrome Display Adapter (MDA) card stores screen memory beginning at 0B0000H. The IBM PC Color Graphics Adapter (CGA) card stores screen memory beginning at 0B8000H. See your computer's reference manual to learn how and where it stores screen memory.

```
(DEFUN SNAPSHOT (ADDR ATOM)
  ((AND (INTEGERP ADDR)
        (<= 0 ADDR (the address space)))
    ((SYMBOLP ATOM)
      (store the bytes in ATOM's print name in
          memory beginning at ADDR)
      T )
    ((AND (INTEGERP ATOM) (<= 0 ATOM 65535))
      (the symbol whose print name consists of
          ATOM bytes of memory beginning at ADDR)
      ) ) )

$ (SETQ *READ-BASE* 16)                    ;Input hexadecimal
16

$ (SETQ PIX (SNAPSHOT 0B0000 200) NIL)     ;Save picture
NIL

$ (SNAPSHOT 0B1000 PIX)                    ;Redisplay picture
T

$ (SETQ *READ-BASE* 0A)                    ;Input decimal
10
```

5.21 CONSOLE SCREEN PRIMITIVES

The console screen primitives facilitate the display and formatting of text and graphics on the console screen. Functions are provided for sizing the console window, positioning the cursor, scrolling text, graphing (points, lines, and circles), and changing video modes, colors, display pages, and cursor lines. Control variables are provided for controlling the blinking and intensity of characters displayed on the screen.

The procedures necessary to accomplish these console screen operations is dependent on the particular computer hardware being used. Therefore when muLISP starts-up, it tries to determine the computer type on which it is running by examining the computer's "finger-print" stored in ROM. If successful, the second line of the muLISP header message (see Chapter 2) displays the computer type that was determined.

However, if muLISP cannot automatically determine the computer type, the prompt

```
1 = Other generic MS-DOS computer
2 = IBM PC or "look-alike" computer
3 = ANSI.SYS screen or VT-100 Terminal
4 = TI Professional Computer
5 = Zenith Z-100 Computer or VT-52 Terminal
6 = Hewlett-Packard HP-150 Computer
7 = Hewlett-Packard HP-110 Computer
8 = NEC Advanced PC or ADM-3A Terminal
9 = NEC PC-9801 Computer
A = Fujitsu Computer

Please enter your computer type number:
```

is displayed. If your computer type is not listed, you can try the various types to see if the console screen functions will work on your computer. If all else fails, choose the "generic MS-DOS computer" type. However, if you select the generic type, most of the primitives described in this section are disabled and simply return NIL.

A muLISP program can also determine the current computer type by examining the byte stored in the muLISP base page at offset 855 decimal (see Appendix B). Thus, the call

```
(CSMEMORY 855)
```

returns the number corresponding to the computer types as listed on the previous page.

In muLISP, the console window is a rectangular region of the console screen to which character and graphics output is confined. The home position is the top left corner of the console window. The base row and base column of the console window are respectively the row and column of the home position *relative* to the full console screen. Initially, the console window is the whole console screen.

Coordinates given to the cursor positioning, scrolling, and graphics functions are *relative* to the home position of the console window. Thus, the coordinates of the home position are row 0 and column 0.

The following discussion assumes a console screen of 25 rows by 80 columns. For other screen sizes simply replace these dimensions by one appropriate for your console screen.

MAKE-WINDOW [*row*, *column*, *rows*, *columns*] Function

If available for your computer type (currently all types but generic MS-DOS), and if *row*, *column*, *rows*, and *columns* are within range, (MAKE-WINDOW *row column rows columns*) establishes a new console window, positions the cursor at the home position, and returns T. The base row of the new console window is *row*, the base column is *column*, the height is *rows*, and the width is *columns*.

row must be zero or a positive integer less than 25. *column* must be zero or a positive integer less than 80. Both *row* and *column* default to 0.

rows must be a positive integer less than or equal to 25 minus *row*. *rows* defaults to 25 minus *row*. *columns* must be a positive integer less than or equal to 80 minus *column*. *columns* defaults to 80 minus *column*.

(MAKE-WINDOW) returns a four-element list consisting of the base row, the base column, the number of rows, and the number of columns in the current console window.

```
$ (MAKE-WINDOW 6 20 12 40)                ;Make 12x40 window
T                                         ;near middle of screen

$ (MAKE-WINDOW)                           ;Current console window
(6 20 12 40)

$ (MAKE-WINDOW NIL NIL NIL NIL)             ;Make the default
T                                         ;full-screen window
```

SET-CURSOR [*row*, *column*] Function
ROW [] Function
COLUMN [] Function

If available for your computer type (currently all types but generic MS-DOS), and if *row* and *column* are zero or positive integers less than the numbers of rows and columns respectively in the current console window, (SET-CURSOR *row column*) repositions the cursor to the specified row and column in the current console window and returns T. Otherwise, SET-CURSOR returns NIL.

(ROW) and (COLUMN) respectively return the row and column position of the cursor relative to the current console window.

```
(DEFUN SET-CURSOR (ROW COL)
  ((AND (<= 0 ROW 24) (<= 0 COL 79))
    (move cursor to ROWth row and COLth column in
        the current console window)
    T ) )

(DEFUN ROW ()
  (cursor row in the current console window) )

(DEFUN COLUMN ()
  (cursor column in the current console window) )

$ (SET-CURSOR 0 0)                    ;Home cursor
T                                     ;Return T
```

CLEAR-SCREEN [] Function
INSERT-LINES [*n*] Function
DELETE-LINES [*n*] Function

If available for your computer type (currently all types but generic MS-DOS), (CLEAR-SCREEN) clears the current console window, moves the cursor to the home position, and returns T. Otherwise, CLEAR-SCREEN returns NIL.

If available for your computer type (currently only the IBM PC), and if *n* is zero or a positive integer, (INSERT-LINES *n*) inserts *n* blank lines in the current console window at the current cursor row and returns T. Otherwise, INSERT-LINES returns NIL. The lines are inserted by scrolling lines down from the current line toward the bottom of the screen.

If available for your computer type (currently only the IBM PC), and if *n* is zero or a positive integer, (DELETE-LINES *n*) deletes *n* lines in the current console window beginning with the current cursor row and returns T. Otherwise, DELETE-LINES returns NIL. The lines are deleted by scrolling lines up from below the cursor.

```
(DEFUN CLEAR-SCREEN ()
  (clear the current console window of text)
  (SET-CURSOR 0 0) )

(DEFUN INSERT-LINES (N)
  ((AND (INTEGERP N) (>= N 0))
    (insert N blank lines at the current cursor
        row in the current console window)
    T ) )

(DEFUN DELETE-LINES (N)
  ((AND (INTEGERP N) (>= N 0))
    (delete N lines beginning with the current
        cursor row in the current console window)
    T ) )

$ (CLEAR-SCREEN)                    ;Clear screen, home cursor
                                    ;Return T
```

FOREGROUND-COLOR [*n*] Function
BACKGROUND-COLOR [*n*] Function

If available for your computer type (currently only the IBM PC), and if *n* is zero or a positive integer less than 256, (FOREGROUND-COLOR *n*) sets the foreground color used to display characters to *n* and returns the previous color. (FOREGROUND-COLOR) returns the current foreground color.

If available for your computer type (currently only the IBM PC), and if *n* is zero or a positive integer less than 256, (BACKGROUND-COLOR *n*) sets the background color used to display characters to *n* and returns the previous color. (BACKGROUND-COLOR) returns the current background color.

If muLISP is running on an IBM PC with a Monochrome Display Adapter (MDA) card, this table summarizes the effect on the display of various settings of the foreground and background color:

Foreground	Background	Effect
0	0	No display
1	0	Underline
7	0	Normal video
0	7	Inverse video

If muLISP is running on an IBM PC with a Color Graphics Adapter (CGA) card or an Enhanced Graphics Adapter (EGA) card, this table summarizes the color associated with each value of *n*:

n	Color	n	Color	n	Color	n	Color
0	Black	4	Red	8	Dark Gray	12	Light Red
1	Blue	5	Magenta	9	Light Blue	13	Light Magenta
2	Green	6	Brown	10	Light Green	14	Yellow
3	Cyan	7	Light Gray	11	Light Cyan	15	White

```
$ (PROGN (BACKGROUND-COLOR 7)          ;Set for inverse video
         (FOREGROUND-COLOR 0))
7
```

PALETTE-COLOR [*n*] Function
BORDER-COLOR [*n*] Function

If available for your computer type (currently only the IBM PC), and if *n* is zero or a positive integer less than 256, (PALETTE-COLOR *n*) attempts to set the palette color number to *n*. If successful, PALETTE-COLOR returns the previous palette color number, otherwise it returns NIL. (PALETTE-COLOR) returns the current palette color number.

If available for your computer type (currently only the IBM PC), and if *n* is zero or a positive integer less than 256, (BORDER-COLOR *n*) sets the border color to *n*. If successful, BORDER-COLOR returns the previous border color, otherwise it returns NIL. (BORDER-COLOR) returns the current border color.

```
$ (PALETTE-COLOR 1)                    ;Use palette number 1
0

$ (BORDER-COLOR 7)                     ;Use border color 7
0
```

VIDEO-MODE [*n*] Functio

If available for your computer type (currently only the IBM PC), and if *n* is zero or a positive integer less than 256, (VIDEO-MODE *n*) attempts to set the video mode to *n*, sets the active display page to the visual display page, makes the console window the full screen, establishes the normal two-line cursor, and clears the screen. If the video mode is successfully set to *n*, VIDEO-MODE returns the previous video mode, otherwise it returns NIL. (VIDEO-MODE) returns the current video mode.

The following table summarizes the IBM PC's video modes (for more detailed information on video modes see the IBM PC Technical Reference Manual):

Mode No.	Mode Type	Colors	Chars	Graphics Resolution	Adapter Card	Monitor Type
0	text	16	40x25	40x25	CGA/EGA	color
1	text	16	40x25	40x25	CGA/EGA	color
2	text	16	80x25	80x25	CGA/EGA	color
3	text	16	80x25	80x25	CGA/EGA	color
4	graphics	4	40x25	320x200	CGA/EGA	color
5	graphics	4	40x25	320x200	CGA/EGA	color
6	graphics	2	80x25	640x200	CGA/EGA	color
7	text	3	80x25	80x25	MDA/EGA	monochrome
8	graphics	16	20x25	160x200	PCjr	color
9	graphics	16	40x25	320x200	PCjr	color
10	graphics	4	80x25	640x200	PCjr	color
13	graphics	16	40x25	320x200	EGA	color
14	graphics	16	80x25	640x200	EGA	color
15	graphics	3	80x25	640x350	EGA	monochrome
16	graphics	16	80x25	640x350	EGA	enhanced

CGA = Color Graphics Adapter
EGA = Enhanced Graphics Adapter
MDA = Monochrome Display Adapter
color = color graphics display monitor
monochrome = monochrome display monitor
enhanced = enhanced graphics display monitor

DISPLAY-PAGE [*n*, *active-only*] Function

DISPLAY-PAGE is only available for IBM PC or compatible computers with a Color Graphics Adapter (CGA) card or an Enhanced Graphics Adapter (EGA) card. Depending on the memory storage requirements of the current video mode, these cards may provide more than one video display page.

The page that is currently being displayed on the console screen is the visual display page. The page that is currently receiving output sent to the console is the active display page. For most muLISP applications the visual and active display pages are the same. However if output has been sent to an active page that is other than the visual page, switching the visual page to the active page "instantly" displays the output on the screen. As shown in the example below, this technique can be used to briefly display "subliminal" messages on the screen.

If *n* is zero or a positive integer less than 256 and *active-only* is NIL or omitted, (DISPLAY-PAGE *n active-only*) attempts to set the active *and* visual display page to *n*. If successful, DISPLAY-PAGE returns the previous visual display page, otherwise it returns NIL.

If *n* is zero or a positive integer less than 256 and *active-only* is nonNIL, (DISPLAY-PAGE *n active-only*) attempts to set the active display page to *n*. If successful, DISPLAY-PAGE returns the previous active display page, otherwise it returns NIL.

(DISPLAY-PAGE) returns the current visual display page. (DISPLAY-PAGE NIL T) returns the current active display page.

```
$ (DEFUN SUBLIMINAL-MESSAGE (MSG)
    (DISPLAY-PAGE 1 T)                ;Switch active page to 1
    (CLEAR-SCREEN)
    (SET-CURSOR 12 (FLOOR (- 80 (LENGTH MSG)) 2))
    (WRITE-STRING MSG)                      ;Write to page 1
    (DISPLAY-PAGE 1)                  ;Switch visual page to 1
    (TONE NIL 100)                        ;Pause 1/10 second
    (DISPLAY-PAGE 0) )                     ;Return to page 0
```

CURSOR-LINES [*start-line*, *end-line*] Function

The console cursor is a blinking block of light on the screen made up of one or more horizontal video lines one column wide. On an IBM PC computer the starting and ending lines of the cursor can be adjusted under software control. If using a monochrome display monitor or an enhanced graphics display monitor, the cursor can consist of up to 14 lines. If using a color graphics display monitor, the cursor can consist of up to 8 lines. In either case, the cursor lines are numbered from the top starting with line 0.

If available for your computer type (currently only the IBM PC), and if *start-line* and *end-line* are non-negative integers less than the maximum number of cursor lines for your monitor, (CURSOR-LINES *start-line end-line*) sets the cursor's start line and end line, and then returns T. Making *end-line* less than *start-line* is legal but it results in some rather bizarre cursor shapes!

(CURSOR-LINES NIL) restores the cursor to the standard, two line cursor shape (lines 11 and 12 for monochrome and enhanced graphics monitors, 6 and 7 for color graphics monitors). When muLISP is terminated, the cursor shape is restored to the shape present at the time muLISP was started.

(CURSOR-LINES) returns a list of the cursor's current start line and end line. These examples assume a monochrome monitor is being used:

```
$ (CURSOR-LINES 0 13)                    ;Make cursor a full block
T

$ (CURSOR-LINES 14 0)                    ;Turn off cursor
T

$ (CURSOR-LINES NIL)                     ;Restore standard shape
T

$ (CURSOR-LINES)                         ;Return standard shape lines
(11 12)
```

READ-DOT [*x*, *y*] — Function
PLOT-DOT [*x*, *y*, *color*] — Function
PLOT-LINE [x_1, y_1, x_2, y_2, *color*] — Function
PLOT-CIRCLE [*x*, *y*, *r*, *color*] — Function

Graphics coordinates are always relative to the home position of the current console window (see MAKE-WINDOW). Thus (0,0) is the coordinate pair of the home position. The maximum value of *x* is one less than the number of character columns in the console window times the width of a character in pixels. The maximum value of *y* is one less than the number of character rows in the console window times the height of a character in pixels. The width and height of a character in pixels for the IBM PC video modes can be determined by dividing the graphics resolution by the character resolution (see VIDEO-MODE).

The number of available colors is determined by the current video mode (see VIDEO-MODE). The colors are numbered from 0 to one less than the number of colors (see FOREGROUND-COLOR). If in a graphics video mode and *color* is greater than or equal to 128, dots are plotted in the color resulting from exclusive-or'ing their original color with *color*.

If (*x*,*y*) is a valid coordinate pair, (READ-DOT *x y*) returns the color of the dot at (*x*,*y*). If *color* is a nonnegative integer less than 256, (PLOT-DOT *x y color*) plots a dot at (*x*,*y*) using color number *color*. If (x_1,y_1) and (x_2,y_2) are valid coordinates, (PLOT-LINE x_1 y_1 x_2 y_2 *color*) plots in *color* a line from (x_1,y_1) to (x_2,y_2). If *r* is a nonnegative integer, (PLOT-CIRCLE *x y r color*) plots in *color* a circle centered at (*x*,*y*) of radius *r*. Except for READ-DOT, these graphics functions are operative even in the text video modes, but at a very low resolution. *color* defaults to 15.

```
$ (DEFUN PLOT-BOX (X1 Y1 X2 Y2 COLOR)
    (PLOT-LINE X1 Y1 X2 Y1 COLOR)
    (PLOT-LINE X2 Y1 X2 Y2 COLOR)
    (PLOT-LINE X2 Y2 X1 Y2 COLOR)
    (PLOT-LINE X1 Y2 X1 Y1 COLOR) )

$ (DEFUN FILL-BOX (X1 Y1 X2 Y2 COLOR)
    (LOOP (PLOT-LINE X1 Y1 X2 Y1 COLOR)
          ((= Y1 Y2))  (INCQ Y1) ) )
```

BLINK: NIL — Control variable
HIGH-INTENSITY: NIL — Control variable

BLINK and *HIGH-INTENSITY* are active only if muLISP is running on an IBM PC or compatible computer in one of the text video modes (see VIDEO-MODE). Otherwise, they will have no effect on characters displayed on the screen.

If *BLINK* is nonNIL, new characters displayed on the screen blink. If *BLINK* is NIL, new characters displayed on the screen do not blink.

If *HIGH-INTENSITY* is nonNIL, new characters are displayed on the screen in high intensity (i.e. bright). If *HIGH-INTENSITY* is NIL, new characters are displayed in low intensity (i.e. dim).

```
$ (SETQ *BLINK* T)
T                                   ;A dim blinking T

$ (SETQ *HIGH-INTENSITY* T)
T                                   ;A bright blinking T

$ (SETQ *BLINK* NIL)
NIL                                 ;A bright steady NIL

$ (SETQ *HIGH-INTENSITY* NIL)
NIL                                 ;A dim steady NIL
```

5.22 MISCELLANEOUS FUNCTIONS

TONE [*frequency*, *duration*] Function

If muLISP is running on an IBM PC, and if *frequency* is an integer in the range 37 to 32767 inclusive and *duration* is a positive integer, (TONE *frequency duration*) generates a tone of *frequency* Hertz (cycles per second) for *duration* milliseconds (thousandths of a second). If muLISP is running on other than an IBM PC or *frequency* is not an integer in the correct range, TONE simply pauses execution for *duration* milliseconds.

When muLISP starts-up, it uses the system's real time clock to adjust TONE so it can generate a tone or pause of the correct duration. Since this can only be an approximation, the duration of the tone or pause generated by TONE may be off by some constant factor.

The following table gives the frequency corresponding to notes for octaves on either side of middle C, which is underlined:

Note	Frequency			
C	131	262	523	1046
D	147	294	587	1175
E	165	330	659	1318
F	175	349	698	1397
G	196	392	784	1568
A	220	440	880	1760
B	247	494	988	1975

```
(DEFUN TONE (FREQ DURATION)
  ((AND (INTEGERP DURATION) (>= DURATION 0))
    ((AND (INTEGERP FREQ) (<= 37 FREQ 32767))
      (generate FREQ Hertz tone for DURATION ms) )
    (pause for DURATION ms) ) )
```

```
$ (PROGN                              ;Generate a familiar movie theme
    (TONE 784 600) (TONE 880 600) (TONE 698 600)
    (TONE 349 600) (TONE 523 900) (TONE NIL 600))
```

TIME [*flag*] Function
GET-DECODED-TIME [] Function

(TIME) returns the elapsed time since the system clock was last reset, expressed in 100ths of a second. If *flag* is nonNIL, (TIME *flag*) resets the clock to 0.

TIME can be used to bench mark programs and help determine the most efficient algorithm to accomplish a particular task. Resetting the time counter to zero does not affect the operating system's time counter, only an internal muLISP time counter.

(GET-DECODED-TIME) returns the current time as a list of six integers signifying the following: year (A.D.), month (1-12), day (1-31), hour (0-23), minute (0-59), and second (0-59).

```
(DEFUN TIME (RESET)
   ((NULL RESET)
     (return the time in 100ths of a second) )
   (set the time counter to time 0)
   0 )
```

```
$ (DEFMACRO BENCHMARK (FORM)              ;Benchmark macro
    (LIST 'PROGN '(TIME T) FORM '(TIME)) )
BENCHMARK

$ (BENCHMARK (RECLAIM))                    ;Compute time for GC
50                                         ;Requires 50 centiseconds

$ (GET-DECODED-TIME)                       ;The 1987 New Year
(1987 1 1 0 0 0)
```

ETSET [*parameter*] Function

(ETSET *parameter*) returns the symbol whose print name is the MS-DOS environment parameter associated with *parameter*. If the parameter not found, GETSET returns NIL.

```
(GETSET 'PATH)                    ;Get MS-DOS search directories
C:\\;C:\\BIN|                     ;They are C:\ and C:\BIN

(GETSET 'COMSPEC)                 ;Get MS-DOS command processor
C:\\COMMAND.COM|                  ;It is C:\COMMAND.COM
```

6. THE muLISP LIBRARY

muLISP includes a library of optional packages that complement the facilities provided by the basic muLISP primitives described in Chapter 5. Packages are provided for implementing arrays, structures, flavors, turtle graphics, mouse interfaces, and dynamic closures.

Each package is distributed as a source file that contains the functions, macros, and variables required for the package. If *filename* specifies the drive, directory, and name of a package's source file, the muLISP command

(LOAD '*filename*)

loads the package and makes its facilities available for use.

This chapter describes the functions, macros, and variables provided by each package. The presentation is similar to that used in Chapter 5, except that the formal definitions of functions and macros are *not* given. Instead you can use the muLISP Editor to examine the actual definitions of the functions, macros, and variables in the package's source file.

6.1 THE ARRAY PACKAGE

The source file ARRAY.LSP provides a set of functions and macros for creating and accessing multi-dimensional arrays in muLISP. This package is patterned after the Common LISP array facility described in Chapter 17 of Common LISP, The Language [Steele, 1984].

Arrays are realized as 3-element lists of the form

(ARRAY *contents dimensions*)

where *dimensions* is a list of the dimensions (i.e. the positive integer lengths of the axes) of the array, and *contents* is a tree whose list structure corresponds to the dimensions. Explicitly including dimension information in arrays provides for fast array bounds checking.

VECTOR [*object*$_1$, ..., *object*$_n$] Function
MAKE-ARRAY [*dimensions*, *option*$_1$, ..., *option*$_n$] Function

(VECTOR *object*$_1$ ... *object*$_n$) creates a vector (i.e. a one dimensional array) whose elements are the objects *object*$_1$ through *object*$_n$ respectively.

If *dimensions* is a list of zero or more positive integers, (MAKE-ARRAY *dimensions*) creates an array whose dimensions are the elements of *dimensions* and whose rank is the length of *dimensions*. If *dimensions* is a positive integer, (MAKE-ARRAY *dimensions*) creates a one-dimensional array whose dimension is *dimensions*. Note that muLISP requires that dimensions be positive, whereas Common LISP only requires that they be non-negative. If *dimensions* is neither a positive integer nor a list of positive integers, a "Dimension Error" error break occurs. The following array options are supported:

:INITIAL-ELEMENT *expression* This option sets the initial value of each array element to *expression*. If this option is omitted, the initial value of each array element is NIL.

:INITIAL-CONTENTS *expression* This option sets the initial value of the array elements to the corresponding elements of *expression* which must be compatible in length with *dimensions* (i.e. the length of *expression* must be equal to the car of *dimensions* and each element of *expression* must be compatible in length with the cdr of *dimensions*). If not compatible in length, a "Dimension Error" error break occurs. If this option is omitted, the initial value of the array elements are determined by the :INITIAL-ELEMENT option.

```
$ (VECTOR 'MERCURY 'VENUS 'EARTH 'MARS)
(ARRAY (MERCURY VENUS EARTH MARS) (4))

$ (MAKE-ARRAY '(2 3) :INITIAL-ELEMENT 5)
(ARRAY ((5 5 5) (5 5 5)) (2 3))

$ (MAKE-ARRAY '(2 3)
              :INITIAL-CONTENTS '((1 2 3)(4 5 6)))
(ARRAY ((1 2 3) (4 5 6)) (2 3))
```

AREF [*array*, *subscript*$_1$, ..., *subscript*$_n$] Ma

If *array* is an *n*-dimensional array and *subscript*$_1$ through *subscript*$_n$ are non-negative integers, (AREF *array subscript*$_1$ ... *subscript*$_n$) returns the specified element of *array*.

(SETF (AREF *array subscript*$_1$ ... *subscript*$_n$) *expression*) sets the specified element of array to *expression*.

For maximum efficiency, AREF does *not* check subscripts to insure that they are within the bounds of the array. If such checking is necessary use the function ARRAY-IN-BOUNDS-P before calling AREF.

```
$ (SETQ PLANETS
        (VECTOR 'MERCURY 'VENUS 'EARTH 'MARS))
(ARRAY (MERCURY VENUS EARTH MARS) (4))

$ (AREF PLANETS 2)
EARTH

$ (AREF PLANETS 4)
NIL
```

ARRAY-RANK [*array*] Function
ARRAY-DIMENSION [*array*, *axis*] Function
ARRAY-DIMENSIONS [*array*] Function
ARRAY-TOTAL-SIZE [*array*] Function

If *array* is an array, (ARRAY-RANK *array*) returns the rank (i.e. the number of dimensions or axes) of *array* as a non-negative integer. Otherwise, ARRAY-RANK returns NIL.

If *array* is an array and *axis* is a non-negative integer less than the rank of *array*, (ARRAY-DIMENSION *array axis*) returns the dimension (i.e. the positive integer length) of dimension number *axis* of *array*. Otherwise, ARRAY-DIMENSION returns NIL. Note that dimension number 0 is the first axis of *array*.

If *array* is an array, (ARRAY-DIMENSIONS *array*) returns the dimensions of *array* as a list of positive integers. Otherwise, ARRAY-DIMENSIONS returns NIL.

If *array* is an array, (ARRAY-TOTAL-SIZE *array*) returns the total number of elements of *array* (i.e. the product of all the dimensions). Otherwise, ARRAY-TOTAL-SIZE returns NIL.

```
$ (SETQ EG (MAKE-ARRAY '(2 3)
             :INITIAL-CONTENTS '((A B C) (D E F))))

$ (ARRAY-RANK EG)
2

$ (ARRAY-DIMENSION EG 1)
3

$ (ARRAY-DIMENSIONS EG)
(2 3)

$ (ARRAY-TOTAL-SIZE EG)
6
```

ARRAYP [*object*] Function
ARRAY-IN-BOUNDS-P [*array*, $subscript_1$, ..., $subscript_n$] Function

If *object* is an array, (ARRAYP *object*) returns T, otherwise it returns NIL. Note that arrays are simply lists whose car is the symbol ARRAY.

If n is the rank of *array* and $subscript_1$ through $subscript_n$ are valid subscripts for the axes of *array*, (ARRAY-IN-BOUNDS-P *array* $subscript_1$ ... $subscript_n$) returns T, otherwise it returns NIL. $subscript_i$ is a valid subscript for the ith axis if it is a non-negative integer less than the dimension of the axis.

```
$ (SETQ EG (MAKE-ARRAY '(2 3)
             :INITIAL-CONTENTS '((A B C) (D E F))))

$ (ARRAYP EG)
T

$ (ARRAYP '(2 3 5))
NIL

$ (ARRAY-IN-BOUNDS-P EG 1 2)
T

$ (ARRAY-IN-BOUNDS-P EG 1 3)
NIL
```

6.2 THE STRUCTURE PACKAGE

The source file STRUCTUR.LSP provides a DEFSTRUCT macro for muLISP. DEFSTRUCT is used to create named and unnamed record structures (i.e. data types) with named components. When a structure is defined, a full set of constructors, accessors, recognizers, copiers, and assignment constructs are created for the new data type. This package is patterned after the Common LISP structure facility described in Chapter 19 of Common LISP, The Language [Steele, 1984].

Named structures are realized as lists of the form

$$(name\ slot_1\ slot_2\ \ldots\ slot_n)$$

and unnamed structures are realized as list of the form

$$(slot_1\ slot_2\ \ldots\ slot_n).$$

All of a structure's functions (i.e. its constructors, accessors, copiers, and predicates) are implemented as macros. This gives good execution speed and integrates nicely with muLISP's SETF implementation. Thus to achieve maximum efficiency, a structure should be defined *before* defining functions that call the structure's functions, so that the macro expansion can occur at compile time rather than at run time. However during program development the control variable MACROEXPAND (see Chapter 5) should be set to NIL to prevent macro expansion until run time so that structures can be redefined.

To achieve maximum efficiency, the structure functions do *not* perform compile time or run time error checking. If such checking is necessary, it can done by the functions that call the structure's functions.

The components of a structure definition (i.e. its options, slot names, slot values, a :NAMED flag, and structure functions) are stored on the property list of the structure's name. In addition, a list of the structures supported is stored on the property list of each structure function.

DEFSTRUCT [(*name option*$_1$ ... *option*$_m$), *slot*$_1$, ..., *slot*$_n$] Macro

If *option*$_1$ through *option*$_m$ are defstruct options and *slot*$_1$ through *slot*$_n$ are slot descriptions, (DEFSTRUCT (*name option*$_1$ ... *option*$_m$) *slot*$_1$ ... *slot*$_n$) creates a new structure named *name*. If only the default defstruct options are required, (DEFSTRUCT *name slot*$_1$ ... *slot*$_n$) can be used instead. Each slot description has the form *slotname* or (*slotname form*), where *slotname* is a symbol and *form* is any muLISP form. Note that slot options are not supported.

If a nonsymbol is given as the name for a structure or a slot in the structure, a "Nonsymbolic Argument" error break occurs. If the definition for a function to be created by DEFSTRUCT is not equal to the existing definition of the function, an "Incompatible Definition" error break occurs. If an unsupported DEFSTRUCT option is given, an "Unsupported Option" error break occurs.

The following defstruct options are supported:

:CONC-NAME This option determines the prefix to be concatenated onto the front of the structure's slot names to make the names of the structure's accessor functions. If this option is given with the nonNIL symbol *prefix* as its argument, *prefix* is used as the accessor function prefix. If this option is not given or given without arguments, the structure's name followed by a hyphen is used as the prefix. If this option is given with NIL as its argument, no prefix is used and the slot names themselves are used for the names of the accessor functions.

:CONSTRUCTOR This option can be included more than once in the option list to make and determine the name of the structure's keyword-driven constructor function, and to make and determine the name of positional constructor functions for the structure.

If this option is given with the nonNIL symbol *cons-name* as its only argument, *cons-name* is used as the name of the structure's keyword-driven constructor function. If this option is not given or given without arguments, (PACK* "MAKE-" *name*) is used as the name of the keyword-driven constructor function. If

given with NIL as its only argument, no keyword-driven constructor function is provided.

If this option is given with the symbol *cons-name* and the list *arglist*, which is a list of the structure's slot names, as its two arguments, a positional constructor function named *cons-name* is provided whose arguments initialize the slots of the structure it creates according to the position of the slots in *arglist*.

:COPIER This option makes and determines the name of the structure's copier function. If this option is given with the nonNIL symbol *copier-name* as its argument, *copier-name* is used as the name of the structure's copier function. If this option is not given or given without arguments, (PACK* "COPY-" *name*) is used as the name of the copier function. If this option is given with NIL as its argument, no copier function is provided.

:PREDICATE This option makes and determines the name of the type predicate for named structures. If this option is given with the nonNIL symbol *predicate-name* as its argument, *predicate-name* is used as the name of the structure's type predicate. If this option is not given or given without arguments, (PACK* *name* "-P") is used as the name of the type predicate. If this option is given with NIL as its argument, no type predicate is provided.

If this option is given for an unnamed structure, an "Unnamed Structure" error break occurs.

:INCLUDE This option includes in the structure being defined the slots of a previously defined structure. In other words, the structure being defined becomes an extension and specialization of the included structure. Note that included structures must be named or unnamed consistently with the structure being defined.

If *structure* is a previously defined structure, (:INCLUDE *structure*) includes the slots of *structure* in the new structure being defined. If $slot_1$ through $slot_n$ are slot descriptions in *structure*, (:INCLUDE *structure* $slot_1$ ... $slot_n$) includes the slots of *structure* in the structure being defined, but with the original default values overridden by $slot_1$ through $slot_n$.

If the name of an undefined structure is given for inclusion in a structure, a "Undefined Structure" error break occurs. If a named structure is given for inclusion in an unnamed structure, or an unnamed structure is given for inclusion in a named structure, an "Include Error" error break occurs.

:TYPE This option used in conjunction with the :NAMED option forces a particular representation for the structure. (:TYPE LIST) causes the structure to be represented as an *unnamed* list, unless the :NAMED option is also used. Note that only the structure type LIST is supported. If this option is omitted, the structure will be represented as a *named* list.

:NAMED This option forces the structure to be named irrespective of any :TYPE specifications. If this option is omitted, the structure will be named unless an explicit (:TYPE LIST) option is specified.

```
$ (DEFSTRUCT PERSON                              ;Define persons
    AGE  SEX  WEIGHT  HEIGHT)
PERSON

$ (SETQ AL (MAKE-PERSON))                        ;Make AL a person

$ (SETF (PERSON-AGE AL) 37)                          ;Al's age
37

$ (DEFSTRUCT (PROGRAMMER (:INCLUDE PERSON))
    LANGUAGES                                    ;Define programmers
    COMPUTERS)                                       ;Include persons
PROGRAMMER

$ (SETQ SAM (MAKE-PROGRAMMER))   ;Make SAM a programmer

$ (SETF (PROGRAMMER-LANGUAGES SAM)               ;Sam's languages
        '(PASCAL C LISP))
(PASCAL C LISP)

$ (SETF (PERSON-SEX SAM) 'MALE)                      ;Sam's sex
MALE
```

6.3 THE FLAVORS PACKAGE

The source file FLAVORS.LSP provides a set of functions and macros to facilitate object-oriented programming in muLISP. An attempt was made to optimize the speed without using excessive memory. Property lists are used to store all the methods and instance variables for the flavors and objects. Most of the common flavors options are included in the package and they follow the convention used on LISP machines. More detailed documentation on flavors can be found in the reference manuals for LISP machines or in Chapter 2 of Hank Bromley's LISP Lore: A Guide to Programming the LISP Machine published by Kluwer Academic Publishers.

A flavor is the general classification of a group of objects. An object is an instance of a flavor. Each flavor has methods that it can respond to and instance variables that can have values. Each object of a specified flavor has the same instance variables although the values associated with each variable may be different. The methods are functions that are called when a message of the same name is sent to the object. Within a method definition variables are locally bound, but only within the context of that method. They do not remain bound when the immediate environment of the method is changed. This behavior is consistent with LISP machines but different from the variable binding of dynamically scoped LISPs, such as muLISP.

Flavors can inherit other flavors along with the methods and instance variables of those inherited flavors. The inherited flavors can also be controlled to some degree by the DEFFLAVOR options :INCLUDED-FLAVORS, :REQUIRED-FLAVORS, and :NO-VANILLA-FLAVOR. The :VANILLA-FLAVOR is included by default in all flavors. It has two methods attached to it although more can be added to conform to the LISP machine environment. The two methods included in this package are the :DESCRIBE and the :WHICH-OPERATIONS methods. When a :DESCRIBE message is sent to an object a description of the object is displayed on the screen. The :WHICH-OPERATIONS message returns a list of all the method names to which the object will respond.

When a message is sent to an object, a method combination is constructed and then evaluated. The default method combination is the DAEMON combination. The inherit flavors are searched top-down.

depth-first for the first method that matches the message. When a matching method is found, it is evaluated and the value returned. All :BEFORE daemons are evaluated before the primary method in the order that the flavors were combined, and all the :AFTER daemons are evaluated in reverse order.

Whoppers are methods that wrap around the primary method combination. All :BEFORE, :AFTER, and primary methods are always surrounded by any whoppers that have been defined for that method with the whopper for the most recent flavor on the inherited flavors list on the outside.

Additionally :AND and :OR method combinations can be declared for each method. An :AND combination gets all the methods that match the message name for an object. These methods are then evaluated in order until one returns NIL or all return a nonNIL value. The :OR combination option works in a similar fashion but returns the first nonNIL value or NIL if all methods return NIL. These options are declared in the DEFFLAVOR function.

When the file FLAVORS.LSP is loaded and if the control variable *FLAVOR-DEMO* is nonNIL, a simple window management system written using the Flavors Package is also loaded. The commented source listing for the window management system shows how to define flavors and methods, and create instances of flavors.

DEFFLAVOR [*name*, *varlist*, *flavorlist*, *option*$_1$, ..., *option*$_n$] Macro

(DEFFLAVOR *name varlist flavorlist option*$_1$ ... *option*$_n$) defines a new flavor named *name* and adds *name* to the global variable *ALL-FLAVOR-NAMES*. *varlist* is a list, each element of which is a variable or a cons of a variable and a list of the default value of that variable. *flavorlist* is a list of all the inherited flavors for the flavor *name*. The following defflavor options are supported:

:INITABLE-INSTANCE-VARIABLES This option makes the instance variables of the flavor initable (i.e. they can be initialized by MAKE-INSTANCE through the use of a keyword). This option is only required for completeness since all instance variables are initable by default.

:GETTABLE-INSTANCE-VARIABLES By itself, this option creates methods for getting the value of *all* the flavor's instance variables. (:GETTABLE-INSTANCE-VARIABLES *var*$_1$ ... *var*$_n$) creates methods for getting the value of only the instance variables *var*$_1$ through *var*$_n$. The name of the retrieval method for the instance variable *var*$_i$ will be :*var*$_i$.

:SETTABLE-INSTANCE-VARIABLES By itself, this option creates methods for setting the values of *all* the flavor's instance variables. (:SETTABLE-INSTANCE-VARIABLES *var*$_1$ ... *var*$_n$) creates methods for setting the value of only the instance variables *var*$_1$ through *var*$_n$. The name of the assignment method for the instance variable *var*$_i$ will be :SET-*var*$_i$.

:REQUIRED-FLAVORS (:REQUIRED-FLAVORS *flavor*$_1$ ... *flavor*$_n$) specifies that any flavor that includes the flavor *name* must also include the flavors *flavor*$_1$ through *flavor*$_n$. If an attempt is made to create an instance of *name* that does not include these flavors, a "Missing Required Flavor" error break occurs.

:INCLUDED-FLAVORS (:INCLUDED-FLAVORS *flavor*$_1$ ... *flavor*$_n$) specifies that any flavor that includes the flavor *name* must also include the flavors *flavor*$_1$ through *flavor*$_n$. If an attempt is made to create an instance of *name* that does not include *flavor*$_i$ where 1 <= *i* <= *n*, then *flavor*$_i$ is added to the inherited flavors of the

instance immediately following the last flavor that has *flavor*$_i$ on its list of included flavors.

METHOD-COMBINATION (:METHOD-COMBINATION (*type order method*$_1$ ... *method*$_n$) ...) declares the way that methods from different flavors are combined when the method is called. *type* can be either :DAEMON (default), :AND, or :OR. *order* is used for the :AND or :OR type and can be either :BASE-FLAVOR-FIRST or :BASE-FLAVOR-LAST.

:NO-VANILLA-FLAVOR This option excludes the primitive methods :DESCRIBE and :WHICH-OPERATIONS for instances of the flavor *name*. This option is rarely used.

```
$ (DEFFLAVOR FOO ((A 4) (B 'CAT) C D) ()
                 :GETTABLE-INSTANCE-VARIABLES
                 :SETTABLE-INSTANCE-VARIABLES)
FOO

$ (DEFFLAVOR BAR () (FOO))
BAR

$ (SETQ THING (MAKE-INSTANCE 'BAR))
BAR-I1

$ (SEND THING :B)
CAT

$ (SEND THING :SET-B 5)
5

$ (SEND THING :B)
5
```

DEFMETHOD [(*flavor type method*), *arglist*, $form_1$, ..., $form_n$] Macro
UNDEFMETHOD [(*flavor type method*)] Macro

If *flavor* is the name of a flavor, (DEFMETHOD (*flavor type method*) *arglist* $form_1$ ... $form_n$) defines the new method *method* for *flavor* of type *type*. *arglist* is a list of variables. *type* is an optional argument that can be either :BEFORE or :AFTER. A method's type determines the order in which the combined methods are evaluated for a given method combination.

When the method is called, the forms are evaluated using an implicit PROGN in the same way as the body of a normal function. Within the forms of the method, instance variables can be referred to directly and SETQ can even be used to change the value of a variable. Note that changing the value of a variable using SETQ does *not* cause :BEFORE and :AFTER daemons to be evaluated. This is true even if they would be evaluated if the value were changed with a :SET-*var* method.

(UNDEFMETHOD (*flavor type method*)) removes the definition of *flavor*'s method *method* of type *type*.

```
$ (DEFFLAVOR FOO ((A 4) (B 'CAT)) ()
                 :GETTABLE-INSTANCE-VARIABLES
                 :SETTABLE-INSTANCE-VARIABLES)
FOO

$ (DEFMETHOD (FOO :BAR) () (LIST A B))
(:METHOD FOO :BAR)

$ (SETQ THING (MAKE-INSTANCE 'FOO))
FOO-I1

$ (SEND THING :BAR)
(4 CAT)

$ (UNDEFMETHOD (FOO :BAR))
T

$ (SEND THING :BAR)
Unclaimed Message Break: (FOO-I1 :BAR)
```

DEFWHOPPER [(*flavor method*), *arglist*, $form_1$, ..., $form_n$] Macro

If *flavor* is the name of a flavor, (DEFWHOPPER (*flavor method*) *arglist* $form_1$ ... $form_n$) defines the new whopper *method* for *flavor*. Whoppers are methods that wrap around the primary method combination. The primary method combination is substituted wherever the function CONTINUE-WHOPPER is called. DEFWHOPPER is similar in all other respects to DEFMETHOD.

```
$ (DEFFLAVOR FOO ((A 4) (B 'CAT)) ()
                         :GETTABLE-INSTANCE-VARIABLES
                         :SETTABLE-INSTANCE-VARIABLES)
FOO

$ (DEFMETHOD (FOO :BAR) (X) (LIST A X))
(:METHOD FOO :BAR)

$ (SETQ THING (MAKE-INSTANCE 'FOO))
FOO-I1

$ (SEND THING :BAR 5)
(4 5)

$ (DEFWHOPPER (FOO :BAR) (X)
                      (CONTINUE-WHOPPER (* 10 X)))
(:METHOD FOO :AROUND :BAR)

$ (SEND THING :BAR 5)
(4 50)
```

MAKE-INSTANCE [*flavor*, *varlist*] Macro

(MAKE-INSTANCE *flavor varlist*) creates an instance of *flavor* and returns it as the symbol formed by concatenating the symbol flavor, the string "-I" and the incremented value of the global variable *INSTANCE-COUNT*. *varlist* consists of alternating keywords and values of the instance variables for *flavor*. After the object has been created it is sent an :init message.

If *flavor* does not include the required flavors, a "Missing Required Flavor" error break occurs. If not all the keywords in *varlist* are handled by the inherited and included flavors of *flavor*, a "Keyword Not Handled" error break occurs.

```
$ (DEFFLAVOR FOO ((A 4) (B 'CAT)) ()
                 :GETTABLE-INSTANCE-VARIABLES
                 :SETTABLE-INSTANCE-VARIABLES)
FOO

$ (DEFMETHOD (FOO :BAR) () (LIST A B))
(:METHOD FOO :BAR)

$ (SETQ THING (MAKE-INSTANCE 'FOO))
FOO-I1

$ (SEND THING :B)
CAT
```

SEND [*object*, *operation*, arg_1, ..., arg_n] Macro
SELF Variable

(SEND *object operation* arg_1 ... arg_n) sends *object* the message or method *operation* with the arguments arg_1 through arg_n. SEND returns the result of the primary operation or the result specified by the method combination given when the *object*'s flavor was defined by DEFFLAVOR. The variable SELF is bound to *object* while the method is being evaluated so the method can manipulate *object* itself.

If *object* has no methods to handle *operation*, an "Unclaimed Message" error break occurs.

```
$ (DEFFLAVOR FOO ((A 4) (B 'CAT)) ()
                 :GETTABLE-INSTANCE-VARIABLES
                 :SETTABLE-INSTANCE-VARIABLES)
FOO

$ (DEFMETHOD (FOO :BAR) () (LIST A B))
(:METHOD FOO :BAR)

$ (SETQ THING (MAKE-INSTANCE 'FOO))
FOO-I1

$ (SEND THING :BAR)
(4 CAT)

$ (SEND THING :SET-A 3)
3

$ (SEND THING :NONSENSE)
Unclaimed Message Break: (FOO-I1 :NONSENSE)
Continue, Break, Abort, Top-level, Restart, Syste
```

6.4 THE TURTLE GRAPHICS PACKAGE

The source file GRAPHICS.LSP provides a set of functions for plotting on the console screen using turtle graphics commands similar to those used in the LOGO computer language.

SIN-DEG [*angle*] Function
COS-DEG [*angle*] Function

If *angle* is a number, (SIN-DEG *angle*) returns the sine of *angle* degrees and (COS-DEG *angle*) returns the cosine of *angle* degrees.

```
$ (SIN-DEG 45)
0.7071067

$ (COS-DEG 0)
1
```

DOT [*x*, *y*] — Function
LINE [x_1, y_1, x_2, y_2] — Function
COLOR — Control variable

If (*x*,*y*) is a valid turtle graphics coordinate, (DOT *x y*) plots a dot at (*x*,*y*) in the color determined by *COLOR*. If (x_1,y_1) and (x_2,y_2) are valid turtle graphics coordinates, (LINE x_1 y_1 x_2 y_2) draws a line from (x_1,y_1) to (x_2,y_2) in the color determined by *COLOR*.

The discussion on the following page includes a table that gives a list of the colors available for each video mode. The color displayed for a given nonnegative integer value of *COLOR* is the *COLOR*th color in the list of colors for the current video mode. Note that the first color of the list is the *zeroth* color.

```
$ (DOT 0 0)                          ;Plot a point
NONE

$ (LINE 0 0 30 50)                   ;Draw a line
NONE
```

SETPOS [*x*, *y*] Function
SETHEADING [*angle*] Function
SETCOLOR [*color*] Function
PENDOWN [] Function
PENUP [] Function
CLR [] Function

If (*x*,*y*) is a valid turtle graphics coordinate, (SETPOS *x y*) positions the turtle at (*x*,*y*). SP can be used as an abbreviation for SETPOS.

If *angle* is a number, (SETHEADING *angle*) points the turtle to a heading of *angle* degrees. SH can be used as an abbreviation for SETHEADING.

If *color* is a valid color for the current video mode, (SETCOLOR *color*) sets the color of the turtle's track to *color*. SC can be used as an abbreviation for SETCOLOR. This table summarizes the valid colors for the video modes:

Mode	Colors
0, 1, 2, 3	black blue green cyan red magenta yellow white
4, 5	black green red white
6	black white
7, 15	black, gray, white
13, 14, 16	black blue green cyan red magenta yellow white

(PENDOWN) causes the track of the turtle to be plotted. (PENUP) causes the track to *not* be plotted. PD and PU can be used as abbreviations for PENDOWN and PENUP respectively.

(CLR) clears the screen and restores the turtle to its initial state: pendown, track color white, heading 0 degrees, at position (0,0).

RIGHT [*angle*] Function
LEFT [*angle*] Function
FORWARD [*distance*] Function
BACK [*distance*] Function

If *angle* is a number, (RIGHT *angle*) turns the turtle right (clockwise) *angle* degrees and returns the turtle's new heading. If *angle* is a number, (LEFT *angle*) turns the turtle left (counter-clockwise) *angle* degrees and returns the turtle's new heading. If *angle* is negative, the turtle is turned *-angle* degrees in the opposite direction. RT and LT can be used as abbreviations for RIGHT and LEFT respectively.

If *distance* is a number, (FORWARD *distance*) moves the turtle forward *distance* pixels and returns the symbol NONE. If *distance* is a number, (BACK *distance*) moves the turtle backward *distance* pixels and returns the symbol NONE. If *distance* is negative, the turtle is moved *-distance* pixels in the opposite direction. Note that FORWARD and BACK perform the necessary rounding so that the turtle ends up exactly on a pixel. FD and BK can be used as abbreviations for FORWARD and BACK respectively.

```
$ (RIGHT 45)                    ;Turn right 45 degrees
45

$ (LT -45)                      ;Turn right 45 degrees
90

$ (FORWARD 50)                  ;Move forward 50 pixels
NONE

$ (BK -50)                      ;Move forward 50 pixels
NONE
```

TURTLE [] Function
TURTLE-WINDOW [] Function

(TURTLE) initializes the turtle parameters to their default values and starts the turtle graphics read-eval-draw loop. The loop first displays the prompt "Command:" in a four line input window at the bottom of the screen. Next it waits for the user to enter a turtle graphics or standard LISP command. Finally it executes the command in the turtle graphics window at the top of the screen. This loop is repeated until the user enters the command "QUIT".

TURTLE-WINDOW is a helper function that creates the appropriate turtle graphics window for the current video mode.

The following sample functions are defined in the GRAPHICS.LSP. They provide examples that use the muLISP turtle graphics functions.

(POLY *side angle*) draws a polygon whose sides are *side* pixels long and whose turn angle is *angle* degrees.

(CORNER-POLY *side angle*) recursively draws polygons that have similar half-sized polygons outside at each corner until the sides of the polygons are reduced to a single pixel.

(SPIRAL *side angle decrement*) draws a spiral whose initial side is *side* pixels long, whose turn angle is *angle* degrees, and whose sides are decremented *decrement* pixels after each turn until the sides disappear.

(SPIROLAT *side angle decrement*) draws a spirolateral by repeatedly calling SPIRAL until the figure is closed (i.e. the total amount of turning reaches 0 modulo 360).

(C-CURVE *depth*) draws the popular "C" curve to *depth* levels of recursion.

(D-CURVE *depth*) draws the intricate "Dragon" curve to *depth* levels of recursion.

6.5 THE MOUSE PACKAGE

The source file MOUSE.LSP provides a set of functions for readin the position and button status of the Microsoft Mouse[tm] or equivalen mouse pointing device from within muLISP. The Microsoft Mou e is m' by Microsoft Corporation, 16011 NE 36th Way, Box 97017, Redmond WA, 98073-9717, U.S.A. Before using these functions the mou e devi driver software most be properly installed (see the Mouse Referen Manual for details).

MOUSE-RESET []

Function

If the mouse hardware and software have been correctly installed, (MOUSE-RESET) resets the mouse parameters to their default values an returns T, otherwise it returns NIL. The mouse should be reset whenever the video mode is changed (see Section 5.21). The default value of the mouse parameters are:

Parameter	Value
Mouse cursor position	screen center (320,100)
Mouse cursor horizontal limits	0 to 639
Mouse cursor vertical limits	0 to 199
Mouse cursor shape (graphics)	arrow
Mouse cursor shape (text)	inverse video box
Mouse cursor display byte	off (-1)
Mouse horizontal aspect ratio	8:8
Mouse vertical aspect ratio	16:8
Light pen emulation	enabled
User-defined call mask	all zeros

```
$ (MOUSE-RESET)                    ;Re et mou
T
```

SHOW-MOUSE [] Function
HIDE-MOUSE [] Function

(SHOW-MOUSE) causes the mouse cursor to be displayed on the console screen and returns NIL. (HIDE-MOUSE) causes the mouse cursor not to be displayed and returns NIL. Note that even when the mouse cursor is not displayed, the mouse software tracks the movement of the mouse.

If the mouse cursor display byte maintained by the mouse software is zero, the mouse cursor is displayed. If the byte is nonzero, the mouse cursor is not displayed. SHOW-MOUSE and HIDE-MOUSE respectively increment and decrement this byte. Thus three consecutive calls to HIDE-MOUSE will require three calls to SHOW-MOUSE before the mouse cursor is redisplayed.

```
$ (SHOW-MOUSE)                    ;Shows mouse
NIL

$ (HIDE-MOUSE)                    ;Hides mouse
NIL
```

MOUSE-POSITION [*x*, *y*] Functio
MOUSE-X Variabl
MOUSE-Y Variabl

If *x* and *y* are non-negative integers less than 640 and 200 respectively (MOUSE-POSITION *x y*) positions the mouse cursor at the coordinat (*x*,*y*) on the screen and returns NIL. (MOUSE-POSITION) sets th variables *MOUSE-X* and *MOUSE-Y* to the current x and coordinates of the mouse cursor and returns the y coordinate.

The mouse uses a coordinate system relative to the upper left corner o the full screen (*not* the home position of the current muLISP window Thus the coordinate (0,0) is the upper left corner of the screen and th coordinate (639,199) is the lower right corner.

```
$ (MOUSE-POSITION 320 100)          ;Center mouse on scree
NIL

$ (MOUSE-POSITION)                          ;y coordinat
100

$ *MOUSE-X*                                 ;x coordinat
320
```

MOUSE-STATUS [] Function
LEFT-BUTTON Variable
RIGHT-BUTTON Variable

(MOUSE-STATUS) sets the variable *LEFT-BUTTON* to T if the left mouse button is currently being pressed, otherwise MOUSE-STATUS sets it to NIL. Also MOUSE-STATUS sets *RIGHT-BUTTON* to the status of the right mouse button. Finally, it returns the new value of *RIGHT-BUTTON*.

```
$ (MOUSE-STATUS)                    ;Check mouse button status
T

$ *LEFT-BUTTON*                     ;Left button not pressed
NIL

$ *RIGHT-BUTTON*                    ;Right button pressed
T
```

BUTTON-PRESSES [*button*] Function
BUTTON-RELEASES [*button*] Function

BUTTON-PRESSES 'LEFT) sets the variables 'MOUSE-X' and 'MOUSE-Y' to the coordinates of the mouse cursor when the left button was last pressed. It returns the number of times the left mouse button has been pressed since BUTTON-PRESSES was last called on the left button. (BUTTON-PRESSES 'RIGHT) does the same thing for the right mouse button. In either case, BUTTON-PRESSES sets the variables 'LEFT-BUTTON' and 'RIGHT-BUTTON' to the current button status (see MOUSE-STATUS).

Similarly, BUTTON-RELEASES sets the variables 'MOUSE-X' and 'MOUSE-Y' to the coordinates of the mouse cursor when the left (right) button was last released. It returns the number of times the left (right) button has been released since BUTTON-RELEASES was last called.

```
$ (BUTTON-PRESSES 'LEFT)          ;Left button pressed 3 times
3

$ (BUTTON-RELEASES 'RIGHT)        ;Right button not released
0
```

MOUSE-BOX [x_1, y_1, x_2, y_2] Function

If (x_1,y_1) and (x_2,y_2) are valid mouse coordinates, (MOUSE-BOX x_1 y_1 x_2 y_2) limits the mouse cursor to the rectangular box defined by these coordinates, moves the mouse cursor into the box, and returns NIL.

Note that even if the mouse cursor has been limited to a portion of the screen, the coordinates returned by the function MOUSE-POSITION are still relative to the full screen.

```
$ (MOUSE-BOX 300 90 340 110)      ;Limit mouse to small box
NIL                               ;in the center of screen

$ (MOUSE-POSITION)                ;Mouse is in upper
90                                ;left corner of box

$ *MOUSE-X*
300
```

6.6 THE CLOSURE PACKAGE

The source file CLOSURE.LSP provides a set of functions for creating and manipulating dynamic closures in muLISP. This package is patterned after the Lisp Machine LISP dynamic closure package.

CLOSURE [*varlist*, *function*] Function

If *varlist* is a list of variables and *function* is a lambda expression or the name of a function, (CLOSURE *varlist function*) creates and returns a dynamic closure of *function* that saves the current values of the variables in *varlist*. Normally a dynamic closure created by CLOSURE is assigned to a variable so the closure can be called using FUNCALL or APPLY.

When a dynamic closure is called, it temporarily binds the variables in *varlist* to the values saved in the closure. Next, it applies *function* to the variables in *varlist*. Finally, it saves the new values of the variables after the function application, restores the variables to their original value, and returns the value of the function application.

Note that dynamic closures can be created manually by a user at the muLISP command level or automatically by a function within a program.

A generator is a special kind of function that produces the successive elements of a sequence. For instance, generators can be defined to produce random numbers or the elements of a sequence of numbers. In order to "know" what element of the sequence to produce next, a generator must keep information about its current state. Implementing a generator as a dynamic closure keeps the state information *local* to the generator. This avoids the possibility of name conflicts that can arise if global variables are used to store the state information.

The dynamic closure FIB generates the successive elements of the Fibonacci Series defined as follows:

$$F(0) = 1$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

```
$ (SETQ F1 1  F2 1)
1

$ (SETQ FIB (CLOSURE '(F1 F2) '(LAMBDA ()
              (PSETQ F2 F1  F1 (+ F1 F2))) ))

$ (FUNCALL FIB)                    ;Generate the Fibonacci Numbers
2

$ (FUNCALL FIB)
3

$ (FUNCALL FIB)
5

$ (FUNCALL FIB)
8

$ (FUNCALL FIB)
13
```

LET-CLOSED [*letlist*, *function*] Mu

If *letlist* is a list of *m* elements of the form (var_1 $value_1$) and *function* the name of a function or a lambda expression, (LET-CLOSED *letlist function*) evaluates the expressions $value_1$ through $value_m$ in order, and temporarily binds the resulting values to the variables var_1 through var_m. Next LET-FORM creates a dynamic closure of *function* over the variables var_1 through var_m with initial values $value_1$ through $value_m$, restores the original values to the variables var_1 through var_m, and returns the dynamic closure. If an element of *letlist* is just the symbol *var*, it is equivalent to the form (*var* NIL) being an element of *letlist*.

Often LET-CLOSED is more convenient than CLOSURE for creating dynamic closures since LET-CLOSED temporarily binds closure variables to an initial value when creating the closure.

```
$ (SETQ FIB (LET-CLOSED ((F1 1) (F2 1))
      '(LAMBDA () (PSETQ F2 F1  F1 (+ F1 F2))) ))

$ (FUNCALL FIB)                    ;Generate the Fibonacci Numbers
2

$ (FUNCALL FIB)
3

$ (FUNCALL FIB)
5

$ (FUNCALL FIB)
8

$ (FUNCALL FIB)
13
```

COPY-CLOSURE [*closure*] Function
CLOSUREP [*object*] Function

If *closure* is a dynamic closure, (COPY-CLOSURE *closure*) creates and returns a copy of *closure*. The copy has the same closure function and variables as *closure* and initially it has the same closure values as *closure*, but it maintains its own list of closure values separate from *closure*'s. Thus, the copy can operate independently of *closure*.

If *object* is a dynamic closure, (CLOSUREP *object*) returns T, otherwise it returns NIL.

```
$ (SETQ FIB1 (LET-CLOSED ((F1 1) (F2 1))
       '(LAMBDA () (PSETQ F2 F1  F1 (+ F1 F2))) ))

$ (FUNCALL FIB1)                                  ;Call FIB1
2

$ (SETQ FIB2 (COPY-CLOSURE FIB1) NIL)             ;Copy FIB1
NIL

$ (FUNCALL FIB2)                                  ;Call FIB2
3

$ (FUNCALL FIB2)                                  ;Call FIB2
5

$ (FUNCALL FIB1)                                  ;FIB1 still at 3
3

$ (CLOSUREP FIB2)                                 ;FIB2 is a closure
T
```

SYMEVAL-IN-CLOSURE [*closure*, *symbol*] Function
SET-IN-CLOSURE [*closure*, *symbol*, *object*] Function

If *closure* is a dynamic closure and *symbol* is a closure variable of *closure*, (SYMEVAL-IN-CLOSURE *closure symbol*) returns the value of *symbol* in *closure*. If *symbol* is not a closure variable of *closure*, SYMEVAL-IN-CLOSURE returns the current value of *symbol*.

If *closure* is a dynamic closure and *symbol* is a closure variable of *closure*, (SET-IN-CLOSURE *closure symbol object*) sets the value of *symbol* in *closure* to *object* and returns *object*. If *symbol* is not a closure variable of *closure*, SET-IN-CLOSURE sets the value of *symbol* to *object* and returns *object*.

```
$ (SETQ F1 1  F2 1)
1

$ (SETQ FIB (CLOSURE '(F1 F2) '(LAMBDA ()
              (PSETQ F2 F1  F1 (+ F1 F2))) ))

$ (FUNCALL FIB)                                    ;Call FIB
2

$ F1                                          ;Value of F1 outside FIB
1

$ (SYMEVAL-IN-CLOSURE FIB 'F1)          ;Value of F1 inside FIB
2

$ (SET-IN-CLOSURE FIB 'F1 5)                 ;Set F1 to 5 in FIB
5

$ (FUNCALL FIB)                                    ;Call FIB
6

$ (SYMEVAL-IN-CLOSURE FIB 'F1)          ;Value of F1 inside FIB
6
```

CLOSURE-ALIST [*closure*] Function
CLOSURE-FUNCTION [*closure*] Function
CLOSURE-VARIABLES [*closure*] Function

If *closure* is a dynamic closure, (CLOSURE-ALIST *closure*) returns a list of the variables of *closure* paired with their closure values. This list is in the form of an alist that is suitable for use by the functions ASSOC and RASSOC.

If *closure* is a dynamic closure, (CLOSURE-FUNCTION *closure*) returns its closure function (i.e. the function name or lambda expression that was given when *closure* was created).

If *closure* is a dynamic closure, (CLOSURE-VARIABLES *closure*) returns a list of its closure variables (i.e. a *copy* of the variable list that was given when *closure* was created).

```
$ (SETQ FIB (LET-CLOSED ((F1 1) (F2 1))
      '(LAMBDA () (PSETQ F2 F1  F1 (+ F1 F2)))))

$ (FUNCALL FIB)                                    ;Call FIB
2

$ (CLOSURE-ALIST FIB)                              ;FIB alist
((F1 . 2) (F2 . 1))

$ (CLOSURE-FUNCTION FIB)                           ;FIB function
(LAMBDA NIL (PSETQ F2 F1 F1 (+ F1 F2)))

$ (CLOSURE-VARIABLES FIB)                          ;FIB variables
(F1 F2)
```

7. THE PROGRAM DEVELOPMENT SYSTEM

muLISP programs can be developed using a standard text edit suitable for your computer and operating system. Program developme using such a nonresident editor consists of repeatedly loading the edito editing the program, saving the program, loading the program int muLISP, debugging the program, reloading the editor, etc. Although yo can use an editor with which you are familiar, this method of progra development is inefficient and cumbersome.

These problems are conveniently resolved by the muLISP progra development system. Its *resident* editor and debugger greatly reduc program development time by taking advantage of the interactive natu of LISP: programs can be created, tested, and debugged all from withi muLISP. This encourages the incremental approach to programming an simplifies and shortens the edit-load-debug-reedit cycle.

The editor's list structure, expression evaluation, matchin parenthesis blinking, and automatic indentation commands make i especially well suited for editing LISP source files. Also, you can choos to make the editor's control key commands either WordStar-like o EMACS-like. If desired, you can even modify the control key command to mimic the behavior of a text editor with which you are alread familiar.

7.1 The muLISP Editor

The muLISP editor is a general purpose source file editor that has been enhanced to facilitate the editing of LISP expressions. It is a screen-oriented editor that is fully integrated into the muLISP window manager. Since it continuously displays a "window" or "picture" of the text in a source file, changes made to the text are seen immediately.

The editor is distributed as the source file EDIT.LSP. If *pathname* specifies the drive and directory containing EDIT.LSP, a muLISP command of the form

(LOAD '*pathname*\\EDIT)

loads the editor along with the file WINDOWS.LSP. While EDIT.LSP is being loaded, the prompt

Make editor WordStar-like or EMACS-like? (W/E)

is displayed and the system waits for you to press W for WordStar or E for EMACS. If you have no preference, we recommend the WordStar-like option. If you plan to use the muLISP debugger, you should also load it at this time (see Section 7.2).

After all the desired window applications have been loaded and the muLISP prompt is displayed, a command of the form

(SAVE '*pathname*\\ED)

saves the editor as the file ED.SYS on the drive and directory specified by *pathname*. Once it has been saved as a SYS file, a DOS command of the form

MULISP *pathname*\ED

loads and runs muLISP with the editor. This time however, since the editor has already been loaded, just press the ESC key to actually start it running.

When the muLISP editor starts running, it clears the screen, draws a border around the edit window, and displays an option menu, a prompt

line, and a status line at the bottom of the screen. When in an edit window, the right end of the status line displays "muLISP Edit". Also displayed on the status line is the name of the file being edited and the row and column position of the cursor.

The general strategy for using the muLISP editor can be summed up in two steps: first, move the cursor to where you want to edit; second, insert, delete, or change the text at that point. The best way to learn your way around the system is to experiment editing (but *not* saving!) the text in an existing source file.

If free memory space becomes a problem, the source file generated using the muLISP editor can be read into muLISP without the editor, hence freeing memory for use by the application program. Also in the current version of the editor, the text being edited is stored entirely in memory. Hence, the editor cannot handle source files larger than about 50 Kilobytes.

Section 7.1.1 describes the options provided by the command mode menu. Section 7.1.2 explains the control key commands available when the editor is in the edit mode. Section 7.1.3 describes how the system stores and manages the text displayed on the screen. This information is helpful if you want to modify or extend the capabilities of the editor.

7.1.1 Command Mode Menu Options

The options described in this subsection are available when the editor is in the command mode. When in this mode, one of the options in the option menu is highlighted in inverse video and the prompt line displays the prompt "Enter option". To switch the editor from edit mode to command mode, press the ESC key.

To select an option when in the command mode, press the space bar or TAB key until the desired option is highlighted, and then press the RETURN key. Note that one letter (normally the first letter) of each option in the option menu is capitalized. Just entering this letter is a faster way to select an option. The multi-word options are selected via a succession of submenus.

Alpha switches the editor from command mode to edit mode. The prompt line displays the prompt "Edit text or press ESC for options" and the cursor moves to the edit window, positioned at the current text entry and editing point. To aid in balancing parentheses, if there is a right parenthesis under or to the left of the cursor, the matching left parenthesis will blink. Similarly, if there is a left parenthesis under or to the right of the cursor, the matching right parenthesis will blink.

Block Begin places the beginning block marker at the current cursor position.

Block Copy copies the text delimited by the beginning and ending block markers and puts the copy at the current cursor position.

Block Delete deletes the text delimited by the beginning and ending block markers.

Block End places the ending block marker at the current cursor position.

Block Jump Begin moves the cursor to the beginning block marker.

Block Jump End moves the cursor to the ending block marker.

Block Move moves the text delimited by the beginning and ending block markers to the current cursor position.

Block Write writes a copy of the text delimited by the beginning and ending block markers to the file you enter following the prompt "BLOCK WRITE file:". If the file already exists, the warning query "Overwrite existing file (Y/N)?" is made to confirm the decision.

Delete Block deletes the text delimited by the beginning and ending block markers.

Delete Character deletes the character under the cursor.

Delete Line deletes the line the cursor is on, including the new line character at the right end of the line.

Delete Right-end deletes the right end of the current line from the cursor to the end of the line but *not* the new line character.

Delete Word deletes the word under and to the right of the cursor.

Jump Bottom moves the cursor to the bottom row of the text window.

Jump End moves the text window and the cursor to the end of the file.

Jump Last moves the cursor to the position it had before the last major cursor movement command.

Jump Start moves the text window and the cursor to the beginning of the file.

Jump Top moves the cursor to the top row of the text window.

List Back moves the cursor backward one list.

List Delete deletes the list under and to the right of the cursor.

List Eval evaluates the expression under and to the right of the cursor. This feature makes it possible to selectively redefine individual function definitions in a file so they can be tested without having to save the entire file on disk and then load it back into muLISP. If there is an open debug window on the screen (see Section 7.2) the result of the expression evaluation is displayed in that window.

List Forward moves the cursor forward one list.

List Next moves the cursor down to the beginning of the line whose first character is neither a blank nor a semicolon (i.e. usually the beginning of the next definition).

Mode sets the text and line editing mode (Insert or Replace) to the mode you select following the prompt "MODE Edit:".

Options Color sets the colors used to display the window frame, window text, option menu, prompt line, status line, background, screen border, and color palette to the colors you specify.

Options Display sets the display adapter (MDA, CGA, EGA, or PCjr), screen mode (Text or Graphics), and screen resolution (Medium or High) to what you specify. If the EGA adapter is selected, the monitor type (Color, Enhanced, or Monochrome) is also set to the type you specify.

Options Execute executes the DOS command you enter following the prompt "Enter DOS command". If you just press the RETURN key without entering a command, muLISP is temporarily suspended and control passes to the DOS command processor. To return to muLISP, you must type EXIT following the DOS prompt.

Options Mute turns on or off the editor's audible signaling of errors when an invalid key is pressed.

Print sends to the computer system's printer the file you enter following the prompt "PRINT file:". You are also prompted for the beginning and ending page numbers. Note that if changes have been made to text in the edit buffer, the buffer must be saved *before* the file is printed so that the changes are reflected in the printout.

Quit abandons the current text, terminates muLISP, and returns control to DOS. If changes have been made to the text since it was last saved, the warning query "Abandon edited text (Y/N)?" is made to confirm the decision.

Replace replaces the first occurrence of the string you enter following the prompt "Enter string to replace" with the string you enter following the prompt "Enter replacement string". The warning query "Replace string (Y/N)?" is made to confirm the decision. If the string to replace is not found, the cursor is moved to the end of the text in the edit buffer.

Search moves the cursor forward to the first occurrence of the string you enter following the prompt "Enter string to find". If the string is not found, the cursor is moved to the end of the text in the edit buffer.

Transfer Clear abandons the text in the edit buffer and clears the text window screen. If changes have been made to the text since it was last saved, the warning query "Abandon edited text (Y/N)?" is made to confirm the decision before abandoning the text.

Transfer Delete deletes from the DOS directory the file you enter following the prompt "TRANSFER DELETE file:".

Transfer Load abandons any existing text in the edit buffer and loads the file you enter following the prompt "TRANSFER LOAD file:". If changes have been made to the existing text since it was last saved, the warning query "Abandon edited text (Y/N)?" is made to confirm the decision.

Transfer Merge merges at the current cursor position the file you enter following the prompt "TRANSFER MERGE file:" with the text already in the edit buffer.

Transfer Save writes the text in the edit buffer to the file you enter following the prompt "TRANSFER SAVE file:". The text in the edit buffer remains unchanged. Used periodically during an extended editing session, this command can prevent the catastrophic loss of edits made to a file in the event of a power failure or software error.

Undelete inserts at the current cursor position the text that was deleted by the most recent multi-character delete command. Single character deletions cannot be undeleted. The Undelete command is useful for restoring text that has been mistakenly deleted. Used in conjunction with a delete command, it is also a convenient way to move or copy a word or line of text.

Window Close closes the window you name following the prompt "WINDOW CLOSE Window:".

Window Designate designates the top pane of the current window to be a window of the type you select following the prompt "WINDOW DESIGNATE Type:".

Window Flip flips to the next pane of the current window. Pressing the F2 function key is an alternative way to flip to the next pane.

Window Goto goes to the window you enter following the prompt "WINDOW GOTO Window:".

Window Next goes to the next window. Pressing the F1 function key is an alternative way to go to the next window.

Window Open opens a new pane of the current window and designates it to be a window of the type you select following the prompt "WINDOW OPEN Type:".

Window Previous goes to the previous window. Pressing the F1 function key while holding down the SHIFT key or CTRL key is an alternative way to go to the previous window.

Window Split Horizontal splits the current window at the line you enter following the prompt "WINDOW SPLIT HORIZONTAL At line:".

Window Split Vertical splits the current window at the column you enter following the prompt "WINDOW SPLIT VERTICAL At column:".

7.1.2 Edit Mode Commands

The muLISP editor can be configured to be either WordStar-like or EMACS-like (see Section 7.1.1). The editor keyboard commands for these two modes are summarized in Appendices E and F respectively. To simplify the following discussion, it is assumed that the editor is WordStar-like.

7.1.2.1 Cursor Control Commands

The cursor control commands are used to position the cursor at the point where the desired change is to be made in the text. The cursor (the block or line of light that shows where the next character will appear when typed) is manipulated primarily using control characters that are entered using your left hand. On most consoles, control characters are entered by pressing the control key, usually marked Ctrl, and typing the desired letter while the control key is still depressed.

The basic cursor controls are oriented so that on a standard keyboard the position of the key corresponds to the direction the cursor is to move. This is illustrated by the following diagram:

```
                 UP
                  E
LEFT    A    S    D    F    RIGHT
                  X
                DOWN
```

These control characters (A, S, D, F, E, and X) move the cursor up, down, left or right from the dot in the center of the diagram. The best way to get the "feel" of these commands is to experiment actually using the muLISP editor on an existing source file.

CTRL-D and CTRL-S respectively move the cursor forward and backward one character in the text.

CTRL-F and CTRL-A respectively move the cursor forward and backward one word, leaving the cursor at the beginning of a word. A word is a string of characters in the text that is delimited by any non-alphanumeric character or the beginning or end of a line.

Note that for the above commands, the text in the edit buffer should be thought of as one long string of text. Thus commands that move the cursor forward, move the cursor down to the beginning of the next line when the right end of the current line is reached. Commands that move the cursor backward, move the cursor up to the end of the previous line when the left end of the current line is reached.

CTRL-E and CTRL-X respectively move the cursor up and down one line.

CTRL-I and TAB move the cursor right to the next tab stop (i.e. the next cursor column that is a multiple of 8) when the editor is in Replace mode. See Section 7.1.2.3 for the effect of TAB when the editor is in the Insert mode.

CTRL-M and RETURN move the cursor down to the beginning of the next line when the editor is in Replace mode. See Section 7.1.2.3 for the effect of RETURN when the editor is in the Insert mode.

CTRL-J and LINEFEED move the cursor to the beginning of the text on the line below the cursor if there is text on the line below the cursor. Otherwise, these commands move the cursor down a line and automatically indent the cursor based on the number of opening and closing parentheses on the original cursor line.

Less frequently used keyboard commands are initiated by entering two-character commands. Generally two-character commands that begin with CTRL-Q "amplify" or "reverse" the effect that the second character would have by itself.

CTRL-Q D and CTRL-Q S respectively move the cursor to the right and left end of the current line.

CTRL-Q X and CTRL-Q E respectively move the cursor to the bottom and top row of the screen.

CTRL-Q I moves the cursor left to the previous tab stop.

CTRL-Q P moves the cursor to the position it had before the last major cursor movement command (e.g. commands such as CTRL-Q X or CTRL-Q E).

CTRL-K 0, CTRL-K 1, CTRL-K 2, and CTRL-K 3 place a correspondingly numbered marker at the current cursor position.

CTRL-Q 0, CTRL-Q 1, CTRL-Q 2, and CTRL-Q 3 move the cursor to the correspondingly numbered marker.

7.1.2.2 Display Control Commands

In general, only part of the text in a file can be shown on the screen at one time. The screen should be thought of as a "window" that displays only a portion of the text. When the editor receives a command that moves the cursor to a position in the text that is not displayed in the window, the text is scrolled up (toward the end of the file) or down (toward the beginning of the file) so the cursor will be displayed.

CTRL-Z scrolls the text up a line so a new line of text will be displayed at the bottom of the screen. Note that this is equivalent to moving the window down a line. The cursor remains at the same position in the text, moving up with the text. However, if the cursor reaches the top of the screen, it remains there. The cursor never leaves the screen.

CTRL-W scrolls the text down a line so a new line of text will be displayed at the top of the screen. Note that this is equivalent to moving the window up a line. The cursor remains at the same position in the text, moving down with the text. However, if the cursor reaches the bottom of the screen, it remains there. The cursor never leaves the screen.

CTRL-R and CTRL-C respectively move the window up and down (i.e. move the text down and up) about 3/4 of a full screen, leaving the cursor at the same position on the screen but moved in the text.

CTRL-Q R and CTRL-Q C respectively move the window and the cursor to the beginning and end of the file.

7.1.2.3 Entering Text Commands

The editor is always in one of two text entry modes. When it is in Replace mode, characters and spaces under and to the right of the cursor will be replaced by new characters as they are entered. This is the easiest way to enter text initially. A mistake can be corrected by simply typing over it with the correction, and text can be "erased" by simply typing over it with spaces.

When the editor is in Insert mode, characters and spaces under and to the right of the cursor are pushed over to the right to make room for the new characters inserted in front of them. When in Insert mode, "Insert" is displayed on the status line at the bottom of the screen (on IBM PC and compatible computers, the cursor height is also doubled).

CTRL-V switches between Insert and Replace modes. Initially, the editor is in Insert mode.

CTRL-I and TAB insert a tab character in the text and moves the cursor right to the next tab stop when the editor is in Insert mode. See Section 7.1.2.1 for the effect of CTRL-I and TAB when the editor is in the Replace mode.

CTRL-M and RETURN insert a new line in the text and moves the cursor down to the beginning of the new line when the editor is in Insert mode. See Section 7.1.2.1 for the effect of RETURN when the editor is in Replace mode.

CTRL-N inserts a new line in the text in either the Insert mode or the Replace mode. However, unlike RETURN, the cursor is not moved and remains at the right end of the newly broken line.

CTRL-P causes the editor to ignore the normal cursor control features of the next character entered, so it can be added to the text like other characters.

7.1.2.4 Deleting Text Commands

In the Replace mode the simplest way to delete text is to simply type spaces or new characters over the text to be deleted. The explicit text deletion commands in this subsection work in both Replace and Insert modes. The block and list structure commands also provide methods for deleting text (see Subsections 7.1.2.5 and 7.1.2.7).

CTRL-G deletes the character under the cursor.

CTRL-H and BACKSPACE delete the character to the immediate left of the cursor.

CTRL-_ and RUBOUT also delete the character to the immediate left of the cursor.

CTRL-T deletes the word to the right of the cursor.

CTRL-Y deletes the line the cursor is on, including the new line character at the end of the line.

CTRL-Q Y deletes the right end of the current line from the cursor to the end of the line but not the new line character.

CTRL-Q RUBOUT deletes the left end of the line from the cursor to the beginning of the line.

CTRL-U restores at the current cursor position the text deleted by the most recent multi-character text deletion command. CTRL-U makes it possible to restore text deleted by mistake and also makes it easy to move or copy a line of text. To *move* a line, place the cursor on the line to be moved, press CTRL-Y to delete the line, move the cursor to the new location for the line, and press CTRL-U to undelete the line. To *copy* a line, press CTRL-U immediately after deleting the line and again after moving the cursor to the new location.

7.1.2.5 Block Commands

The block commands permit a block of text to be moved, copied, deleted, read from or written to a file. Two and three-character commands that begin with CTRL-O initiate the block commands. The second and third character of CTRL-O commands are mnemonic. Alternatively and in accordance with WordStar, two-character commands that begin with CTRL-K can be used to initiate block commands. But the second character of these CTRL-K commands are not all mnemonic.

CTRL-O B and CTRL-K B place the beginning block marker at the current cursor position.

CTRL-O E and CTRL-K K place the ending block marker at the current cursor position.

CTRL-O M and CTRL-K V move the text delimited by the beginning and ending block markers from its present position to the current cursor position.

CTRL-O C and CTRL-K C copy the text delimited by the beginning and ending block markers and puts the copy at the current cursor position.

CTRL-O D and CTRL-K Y delete the text delimited by the beginning and ending block markers.

CTRL-O W and CTRL-K W write a copy of the text delimited by the beginning and ending block markers to the file you enter following the prompt "BLOCK WRITE file:". If the file already exists, the warning query "Overwrite existing file (Y/N)?" is made to confirm the decision.

CTRL-O J B and CTRL-Q B move the cursor to the beginning block marker.

CTRL-O J K and CTRL-Q K move the cursor to the ending block marker.

7.1.2.6 Search and Replace Commands

CTRL-Q F moves the cursor forward to the first occurrence of the string you enter following the prompt "Enter string to replace". If the string is not found, the cursor is moved to the end of the text in the edit buffer.

CTRL-Q A replaces the first occurrence of the string you enter following the prompt "Enter string to replace" with the string you enter following the prompt "Enter replacement string". The warning query "Replace string (Y/N)?" is made to confirm the decision. If the string to replace is not found, the cursor is moved to the end of the text in the edit buffer.

CTRL-L repeats the last search or replace command looking for the *next* occurrence of the string. If repeating a replace command, the warning query "Replace string (Y/N)?" is made to confirm the decision.

CTRL-Q L repeats the last search and replace command looking for *all* occurrences of the string from the current cursor position to the end of the text. The warning query "Replace string (Y/N)?" is made to confirm each decision. Press the ESC key to terminate the multiple replacement command.

CTRL-Q V moves the cursor to the point at which the last search or replace operation was initiated.

7.1.2.7 List Structure Commands

The list structure commands are for editing lists in the text. Two-character commands that begin with ESC initiate the list structure commands. If running muLISP on an IBM PC or compatible computer, ALT key commands may be more convenient.

ESC D and ALT-D move the cursor forward one list.

ESC S and ALT-S move the cursor backward one list.

ESC C and ALT-C move the cursor down to the beginning of the line whose first character is neither a blank nor a semicolon (i.e. usually the beginning of the next definition).

ESC R and ALT-R move the cursor up to the beginning of the line whose first character is neither a blank nor a semicolon.

ESC T and ALT-T delete the list under and to the right of the cursor.

ESC I and ALT-I evaluate the list under and to the right of the cursor. This feature makes it possible to selectively redefine edited function definitions so they can be tested.

7.1.2.8 File Saving Commands

The file saving commands are used to save the text as a file or abandon the text. These commands are initiated by two-character commands beginning with CTRL-K.

CTRL-K D and CTRL-K S write the text in the edit buffer to the file you enter following the prompt "TRANSFER SAVE file:". The text in the edit buffer remains unchanged. Used periodically during an extended editing session, these commands can prevent the catastrophic loss of edits made to a file in the event of a power failure or software error.

CTRL-K A and CTRL-K Q abandon the text in the edit buffer and clears the text window screen. If changes have been made to the text since it was last saved, the warning query "Abandon edited text (Y/N)?" is made to confirm the decision before abandoning the text.

CTRL-K R merges at the current cursor position the file you enter following the prompt "TRANSFER MERGE file:" with the text already in the edit buffer.

7.1.3 Theory of Operation

Since the editor is written entirely in muLISP, you can modify or even add features to the editor to suit your taste. This subsection briefly describes how text is stored in the edit buffer and how control key commands are processed. See the source listing of EDIT.LSP for details.

One of the simplest changes you can make to the editor is to reassign the control keys so the muLISP editor will "feel" like an editor with which you are already familiar. When a control key is pressed, the associated function on the alist bound to the global variable *EDIT-DEMONS* is called to perform the desired operation. Near the end of EDIT.LSP, *EDIT-DEMONS* is assigned to either the WordStar-like alist or the EMACS-like alist based on your choice. You can change the control key commands simply by reassigning *EDIT-DEMONS* to your own alist.

More ambitious changes, such as modifying or adding new keyboard commands to the editor, require a thorough understanding of how the text is stored and how to use the editor's existing text primitives to manipulate the text and change the position of the point. Throughout this discussion, the point is the position in the text between the character the cursor is on and the character to its immediate left.

The text is stored as a list of symbols whose print names are the string of characters for each line of text. *BELOW-POINT* is the list for the text below the point. *ABOVE-POINT* is the list for the text above the point. The first line below and above the point are the first elements of *BELOW-POINT* and *ABOVE-POINT* respectively. The second line is the second element, etc.

POINT-ROW and *POINT-COL* are non-negative integers that store the current row and column of the point relative to the start of the text. *CURSOR-ROW* and *CURSOR-COL* store the current row and column of the point relative to the top left corner of the current edit window.

The function JUMP-TO-POSN should be used to move the point within the text since it automatically updates the above variables and scrolls the screen if necessary. (JUMP-TO-POSN *row column*) moves the point to text row *row* and text column *column*.

This editor variables table summarizes the muLISP editor's variables and their significance.

Variable	Description
ABOVE-POINT	text above the point
BELOW-POINT	text below the point
CURSOR-COL	cursor column position in the edit window
CURSOR-ROW	cursor row position in the edit window
DELETED-TEXT	most recently deleted text
EDIT-MODE	T if in edit mode, NIL if in command mode
END-COL	ending column of the block in the text
END-ROW	ending row of the block in the text
FILE-NAME	file currently being edited
FIND-COL	text column of the last search and replace
FIND-ROW	text row of the last search and replace
FIND-STRG	current "Find string"
INSERT-MODE	T if in the Insert mode, NIL if in Replace mode
LAST-COL	text column of the last significant move
LAST-ROW	text row of the last significant move
POINT-COL	text column of the point
POINT-ROW	text row of the point
REPL-STRG	current "Replace string"
SCREEN	list of print names currently being displayed
START-COL	starting column of the block in the text
START-ROW	starting row of the block in the text
STAT-COL	column number displayed in status window
STAT-ROW	row number being displayed in status window
TEXT-DIRTY	nonNIL if the text has been changed since it was last saved

This editor control variable table summarizes the control variable that you can set. They are initialized near the end of EDIT.LSP.

Constant	Value	Description
DEFAULT-TYPE	LSP	Default file name extension
PRINTER-DEVICE	PRN	Printer device name
CASE-IGNORE	NIL	Ignore case during string searches
USE-TABS	T	Use tab chars when saving files
USE-FORMFEED	NIL	Use formfeed chars when printing

This printer control variable table summarizes the control variable associated with the printer. They are initialized near the end of EDIT.LSP. They may need to be reassigned for your printer.

Constant	Value	Description
PAGE-LINES	66	Page length in lines
PAGE-COLUMNS	85	Page width in columns
TOP-MARGIN	3	Desired top margin (Must be >= *INITIAL-LINE*)
BOTTOM-MARGIN	5	Desired bottom margin
FOOTER-MARGIN	2	Desired footer margin
LEFT-MARGIN	8	Desired left margin (Must be >= *INITIAL-COLUMN*)
RIGHT-MARGIN	NIL	Desired right margin (NIL means truncate long lines)
INITIAL-LINE	2	Printer's initial line number
INITIAL-COLUMN	3	Printer's column after a RETURN
INIT-PRINTER	""	Printer's initialization string
STOP-PRINTER	""	Printer's termination string

7.2 The muLISP Debugger

The muLISP debugger is a powerful tool designed to help you locate bugs (errors) in muLISP programs by displaying calls and returns of traced functions. Tracing functions lets you observe the progress of a program or stop a program at critical points so you can examine and/or modify the program's variables. Fully integrated into the muLISP windowing system, the debugger can display trace output in its own window, separate from the program's output.

The debugger is distributed as the source file DEBUG.LSP. If *pathname* specifies the drive and directory containing DEBUG.LSP, a muLISP command of the form

(LOAD '*pathname*\\DEBUG)

loads the debugger along with the file WINDOWS.LSP. If the muLISP editor is also loaded, the combined package provides a complete program development system.

When the muLISP prompt appears, a muLISP command of the form

(SAVE '*pathname*\\PDS)

saves the program development system as the SYS file PDS.SYS on the drive and directory specified by *pathname*. Once the system has been saved, a DOS command of the form

MULISP *pathname*\PDS

loads and runs both muLISP and the program development system.

When the muLISP program development system starts running, it begins with an edit window (see Section 7.1). Use the Window Open or Window Split command to create a debug window. When in a debug window, the right end of the status line at the bottom of the screen displays "muLISP Debug". Also displayed on the status line is the percentage of *free* space in the data and current code spaces.

Section 7.2.1 describes the options provided by the command mode menu. Section 7.2.2 shows typical trace output.

7.2.1 Command Mode Menu Options

The following options are provided by debug windows. To select an option, press the space bar or TAB key until the desired option is highlighted and then press the RETURN key. Alternatively, just enter the capitalized first letter of the desired option.

Author allows you to enter forms for evaluation just as you do following the muLISP dollar-sign prompt. The input line and result are displayed in the debug window.

Backtrace displays the current nesting of traced function calls. This option is only available during a step mode break (see Section 7.2.2).

Debug Mode sets the debug mode to the one (Continuous, Step, or OFF) selected by you following the prompt "DEBUG MODE: ". Section 7.2.2 describes the effect the debug modes have on function tracing.

Debug Trace traces the functions entered by you following the prompt "DEBUG TRACE function: ". If a primitively defined function or one of the special forms SETQ, INCQ, or DECQ is entered, the system also prompts you for the user-defined functions in which the primitive function is to be traced. The other special forms cannot be traced.

Debug Untrace All untraces or clears all the functions in the current debug window.

Debug Untrace Some untraces or clears the functions entered by you following the prompt "DEBUG UNTRACE function: ". If a primitively defined function is entered, the system also prompts you for the user-defined functions in which the primitive function is to be untraced.

History displays the most recent *HISTORY-LENGTH* lines of trace history even though the debug mode may have been OFF during the trace. Using the Author option, you can change the positive integer value of *HISTORY-LENGTH* as desired. Its initial value is 25.

Mode sets the line editing mode (Insert or Replace) selected by you following the prompt "MODE Edit: ".

Options Color sets the color palette and the colors used to display the window frame, window text, option menu, prompt line, status line, background, and screen border.

Options Display sets the display adapter (MDA, CGA, EGA, or PCjr), screen mode (Text or Graphics), and screen resolution (Medium or High) that you specify. Also if the EGA adapter is selected, this option sets the monitor type (Color, Enhanced, or Monochrome) that you specify.

Options Execute executes the DOS command entered by you following the prompt "Enter DOS command". If the RETURN key is pressed without entering a command, muLISP is temporarily suspended and control passes to the DOS command processor. To return to muLISP, type EXIT following the DOS prompt.

Options Mute lets you choose whether or not errors will be signaled by an audible beep.

Print sends to the computer system's printer the file named by you following the prompt "PRINT file: ". You can also specify the beginning and ending page numbers to print.

Quit abandons the program development system, terminates muLISP, and returns control to DOS. If changes have been made to text being edited in an edit window since it was last saved, the warning query "Abandon edited text (Y/N)?" is made to confirm the decision. This option is *not* available during a step mode break.

Resume proceeds from a step mode break. This option is only available during a step mode break.

Transfer Delete deletes from the MS-DOS directory the file named by you following the prompt "TRANSFER DELETE file: ".

Transfer Load loads the source file named by you following the promp "TRANSFER LOAD source file: ". As each expression in the file i read and evaluated, its value is displayed in the debug window if the control variable *SHOW-LOAD* is nonNIL.

Transfer Save saves the current muLISP environment as the SYS fil named by you following the prompt "TRANSFER SAVE SYS file: "

Window performs the same windowing operations as the editor's windo options (see Section 7.1). These options are *not* available during a step mode break.

7.2.2 Debugger Display

Throughout this discussion of the debugger, we assume that the functions APP and REV have been defined as follows:

```
(DEFUN APP (LST1 LST2)
  ((NULL LST1) LST2)
  (CONS (CAR LST1) (APP (CDR LST1) LST2)) )

(DEFUN REV (LST)
  ((NULL LST) NIL)
  (APP (REV (CDR LST)) (LIST (CAR LST))) )
```

APP and REV are functionally equivalent to the primitive muLISP functions APPEND and REVERSE respectively. Since APP and REV are less general and less efficient than their machine language counterparts, we have given them different names.

To display a continuous trace of function calls and returns, set the debug mode to Continuous. When a traced function is called, the function's name is displayed followed by a list of its arguments. When a traced function returns, its name, an equal sign, and its value are displayed. If REV and APP are traced, the following is displayed during the evaluation of the call (REV '(A B)):

```
REV ((A B))
  REV ((B))
    REV (NIL)
    REV = NIL
    APP (NIL (B))
    APP = (B)
  REV = (B)
  APP ((B) (A))
    APP (NIL (A))
    APP = (A)
  APP = (B A)
REV = (B A)
```

Note how indentation is used to graphically indicate the nesting of function calls. If trace output is scrolling too rapidly to read, use CTRL-S to temporarily freeze and unfreeze the display.

If the control variable *PRINT-TRACE* is nonNIL, trace output i also sent to the system's printer to produce a "hard copy" record of a ba trace. Initially, *PRINT-TRACE* is NIL.

To avoid producing prodigious amounts of useless trace output, s the debug mode to Off. Even in the Off mode, the History optio displays the most recent trace output that had been suppressed. A fixe length buffer is used to store the history. The buffer length is set by th control variable *HISTORY-LENGTH*. Its initial value is 25. It can b changed using the Author option.

To suspend evaluation of a program when a traced function i called, set the debug mode to Step. This lets you examine and/or chang the value of program variables, display a backtrace of the current nestin of traced function calls, and display a trace history. Select the Resum option when you are ready to continue evaluation of the program.

7.3 The muLISP Window Manager

The muLISP window manager provides a flexible, menu-driven interface for window-oriented application programs. The many windowing services the manager furnishes greatly simplify the writing of such application programs. It presents the end-user with a standard interface that is easy to learn and use. The muLISP editor and debugger are application programs that are built on the window manager.

This section describes how to write application programs using the window manager. The simple data entry program ADDRESS.LSP is used to illustrate the basic concepts involved. You should print out a listing of ADDRESS.LSP since we will refer to it throughout the remainder of this section.

The window manager is distributed as the file WINDOWS.LSP. If *pathname* specifies the drive and directory containing WINDOWS.LSP, a muLISP command of the form

(LOAD '*pathname*\\WINDOWS)

loads the window manager. Normally, however, windowing applications automatically chain load WINDOWS.LSP so it is unnecessary to explicitly load the windows manager. Code at the end of ADDRESS.LSP illustrates how to chain load WINDOWS.LSP.

The entry point for the window manager is the function WINDOWS. If you want the window manager to start automatically when it is saved as a SYS, COM, or EXE file, set DRIVER to WINDOWS (see the effect of the control variable DRIVER in Chapter 5).

There can be several windowing applications loaded at one time. The initial window displayed by the window manager is the one assigned to the control variable *INIT-WINDOW*. If you intend for your application to be the initial application, set *INIT-WINDOW* to the name of its top level entry function. The entry function for the data entry program is "Address".

The control variable *WINDOW-TYPES* is a list of the top-level entry functions for each of the windowing applications currently loaded. You should merge the name of your application program's entry function with the existing list of window types. When a new window is being opened, these names are presented to the user so one can be selected. For this reason the name of your application program's top-level entry function should be a capitalized string naming the application.

7.3.1 Window Manager Commands

The top-level entry function of an application is called by the window manager when an instance of the application is needed to perform some operation. The command to perform the operation is passed as the first argument to the entry function. The remaining arguments contain the state information about the instance of the application. The entry function must be able to handle the following commands from the window manager:

CREATE-WINDOW This command directs the entry function to return a list of the state information appropriate for a newly created window. The first element of the list is the application's name. The remaining elements correspond to the values of all but the first argument of the top-level entry function. This command should *not* send output to the screen.

If the window manager is started automatically when it is loaded as a SYS, COM, or EXE file, the global variable *COMMAND-LINE* is set to the MS-DOS command line string. In the code for the CREATE-WINDOW command, the value of *COMMAND-LINE* can be used as the initial file name to process. See ADDRESS.LSP as an example.

CLOSE-WINDOW This command is issued to the entry function when the window manager desires to close an instance of an application. The entry function should return NIL if it is not acceptable to close the window. In the data entry program this is done by the predicate ABANDON-RECORDS which asks the user if it is alright to abandon records that have been changed. This command should use the prompt line to request information from the user and should *not* otherwise send output to the screen.

UPDATE-WINDOW This command directs the entry function to clear and then refresh the display in the working window of an instance of an application. In the data entry program this is done by the function UPDATE-ADDRESS which displays the data entry prompts in dim characters and the current record in bold.

RUN-WINDOW This command directs the entry function to actually run an instance of an application. For the data entry program this involves displaying information on the status line using the function ADDRESS-STATUS. Then the program enters a loop which calls upon DISPLAY-RECORD to display the current record and then calls upon EXECUTE-OPTION to execute an option selected by the user.

7.3.2 Window Manager Utilities

The window manager provides several useful utility functions that simplify the writing of application programs and present the user with a consistent interface. Using the muLISP editor on the source file WINDOWS.LSP you can examine the definitions of the utility functions described in this subsection.

CHANGE-WINDOW [] Function

CHANGE-WINDOW provides the standard submemu for the top-level "Window" option. Facilities are provided for opening, closing, splitting, and changing windows.

CONTINUE-PROMPT [] Function

CONTINUE-PROMPT displays "Press any key to continue" on the prompt line, clears the console input buffer, and waits for the user to press any key other than CTRL-S. It is useful for giving the user a chance to read information displayed on the screen before it is erased.

ERROR-BEEP [] Function

If warning tones are not being muted, ERROR-BEEP clears the console input buffer and sounds a short beep. It is useful for warning the user of an invalid input or some other error condition.

EXECUTE-OPTION [*prompt*, *option-tree*] Function

EXECUTE-OPTION provides a consistent menu-driven interface for application programs. Its first argument is the desired option prompt message. Its second argument is the application program's option tree. An entry in the option tree consists of a capitalized option name dotted with the function to execute when the option is selected. See the assignment to *ADDRESS-OPTIONS* in ADDRESS.LSP for an example of an option tree.

GO-DOS [] Function

GO-DOS lets the user enter an MS-DOS command. Normally GO-DOS is called when an "Execute" option is chosen.

PROMPT-YN [*prompt*] Function

PROMPT-YN displays *prompt* on prompt line and waits for the user to select "Y" or "N". If "Y" is selected, PROMPT-YN returns T. If "N" is selected, PROMPT-YN returns NIL.

QUIT-PROGRAM [] Function

QUIT-PROGRAM requests the window manager to terminate all running applications and return control to MS-DOS. Before doing so, the window manager ensures that all running applications approve of the termination. If some application (perhaps the application that issued the QUIT-PROGRAM request) disapproves, the window manager uses a RUN-WINDOW command to return control to the application that issued the QUIT-PROGRAM request. Normally QUIT-PROGRAM is called when a "Quit" option is chosen.

SET-COLOR [] Function
SET-DISPLAY [] Function
SET-INSERT [] Function
SET-MUTE [] Function

These functions provide a range of services that are normally included in the submenu for the "Options" top-level option. SET-COLOR lets the user choose the colors to use for the working window frame, the working window contents, the option window text, the prompt line text, the status line text, and the whole screen's background. SET-DISPLAY lets the user choose the video display mode. SET-INSERT lets the user select either the "insert" or "replace" mode of input. SET-MUTE lets the user mute warning messages.

SHOW-PROMPT [*prompt*] Function

SHOW-PROMPT displays *prompt* on the prompt line and updates the variable *'PROMPT** which stores what is currently being displayed in the prompt line.

UPDATE-STATE [*arglist*] Function

If *arglist* is a list of an application's state variables, (UPDATE-STATE *arglist*) updates the values of the variables in the global variable *'WINDOWS**. UPDATE-STATE should be the last function called by the top-level entry function if there is any possibility that the state has changed since the entry function was called by the window manager. In ADDRESS.LSP the call to UPDATE-STATE is enclosed in an UNWIND-PROTECT form after the RUN-WINDOW loop. The call is protected in this way since the loop is exited by means of a throw.

WINDOWS [*command-line*] Function

WINDOWS is the entry point for the window manager. After performing some initialization and displaying the borders for a single window, WINDOWS runs the initial application. When an application terminates by means of a throw, the value of *'THROW-TAG** is used to determine the action that WINDOWS takes.

8. THE muLISP COMPILER*

The incremental muLISP compiler can greatly improve the execution speed of application programs by transforming their function definitions into native machine code. With the exception of faster execution, the behavior of a function does not change after it is compiled: it can be be invoked interactively at the muLISP driver level, it can be used in the definitions of other functions, it can be applied indirectly through APPLY, FUNCALL, or the mapping functions, and so forth.

After compiling the functions in a program, the function DELETE-COMPILER can be used to regain most of the memory space used by the compiler. Finally, the function SAVE can be used to save the program, including the compiled machine code, as a fast loading SYS file or COM file (see Section 2.6 and Section 5.15.4).

The compiler is fully incremental in the sense that compiled functions call the most recent definition of user-defined functions. Thus, functions can be compiled one at a time and in any order. Moreover, previously compiled functions can be redefined (and then recompiled if desired), and the program can be rerun *without* having to recompile the functions that called them.

At any point in the development of a program, the compiler can be loaded into muLISP and used as an incremental compiler. The function COMPILE is used to individually compile, or recompile, user-defined functions in the program.

Alternatively, the compiler can be used as a file compiler. The function COMPILE-FILE is used to automatically compile all the functions in a file, or list of files, and then delete the compiler. Although file compilation is not as flexible for program development as incremental compilation, it results in the most compact code and the most thorough elimination of the compiler from the muLISP data spaces.

* The muLISP Compiler is an optional package that can be purchased from Soft Warehouse, Inc.

The muLISP compiler is distributed as five files: COMPILE.LSP, COMPILE0.LSP, COMPILE1.LSP, COMPILE2.LSP, and COMPILE3.LSP. To load the compiler into muLISP, issue a command of the form

(LOAD "*pathname*\\COMPILE")

where *pathname* specifies the drive and directory of the five compiler files. After a few seconds, the system will ask if you want to use the compiler as an incremental compiler or a file compiler.

If you choose to run as an incremental compiler, the entire compiler is loaded and the functions COMPILE and DELETE-COMPILER (see Section 8.1) are available for use. Although the compiler can be loaded at any point in the development of a muLISP program, we recommend loading the compiler *after* the functions to be compiled are defined. This maximizes the amount of memory regained when the compiler is deleted.

If you choose to run as a file compiler, only the function COMPILE-FILE is loaded (see Section 8.1). When COMPILE-FILE is subsequently called, it automatically loads the remainder of the compiler *after* the file (or list of files) being compiled has been loaded. This maximizes the amount of memory regained when the compiler is deleted.

8.1 COMPILER FUNCTIONS

This section documents the functions that make up the user interface to the muLISP compiler. Their use is described in the remainder of this chapter.

COMPILE [*name*] Function

If *name* is a user-defined eval function, (COMPILE *name*) translates the definition of *name* into assembly language instructions, assembles the instructions into 8086 machine code, loads the machine code into the muLISP code segment, and redefines *name* to invoke the loaded machine code when called.

If *name* is not a user-defined eval function, COMPILE generates a "Non-eval Function" error break. No-eval functions, macro functions, and muLISP primitives can *not* be compiled.

However, the definition of *name* may contain any legitimate muLISP form including:

* symbols;
* numbers;
* calls to primitive muLISP functions and special forms;
* calls to user-defined functions of any type (LAMBDA or NLAMBDA, spread or no-spread, compiled or uncompiled);
* macro constructions;
* application of LAMBDA or NLAMBDA bodies (such as are produced by the LET macro); and,
* implicit CONDs and PROGNs.

All user-defined functions called directly in the definition of *name* must be defined, *though not necessarily compiled*, so the compiler can determine their type (LAMBDA, NLAMBDA, or MACRO).

If COMPILE encounters no errors, it returns a list consisting of *name*, the number of bytes of machine code needed to compile *name*, and the entry point to the machine code as an offset in the muLISP code segment. If the compiler detects an error, a diagnostic error message is displayed and an error break occurs. *name* is *not* redefined if an error break occurs.

The following example shows how the user-defined function RCONS is defined, compiled and then run. The code length and entry point returned by COMPILE will vary.

```
$ (DEFUN RCONS (X Y)                          ;Define RCONS
    (CONS Y X) )
RCONS

$ (RCONS 1 2)                                 ;Test RCONS
(2 . 1)

$ (GETD 'RCONS)                    ;LAMBDA definition of RCONS
(LAMBDA (X Y) (CONS Y X))

$ (COMPILE 'RCONS)                            ;Compile RCONS
(RCONS 62 37457)

$ (GETD 'RCONS)                    ;Machine definition of RCONS
37457

$ (RCONS 1 2)                                 ;Retest RCONS
(2 . 1)
```

DELETE-COMPILER [] Macro

(DELETE-COMPILER) deletes the compiler from the muLISP environment by removing the definitions of all compiler functions, as well as the values and properties of certain compiler symbols. It returns the symbol DELETE-COMPILER, which is the result of removing its own definition using the function REMD.

The compiler takes up a significant amount of memory space in the muLISP environment. The function DELETE-COMPILER can be used to free this memory space when you have finished compiling functions. Deleting the compiler enables the muLISP garbage collector to reclaim most of the memory space used by the compiler's function definitions, symbols, and properties.

When the muLISP compiler is used as a file compiler, COMPILE-FILE calls DELETE-COMPILER automatically, making it unnecessary for the user to do so.

```
$ (RECLAIM)                          ;Free bytes with compiler
311600

$ (DELETE-COMPILER)                  ;Delete compiler
DELETE-COMPILER

$ (RECLAIM)                          ;Free bytes w/o compiler
337767

$ (- * ***)                          ;Bytes reclaimed
26167
```

COMPILE-FILE [*compile-files*, *load-files*] Function

If *compile-files* and *load-files* are file names *or* lists of file names, (COMPILE-FILE *compile-files load-files*) loads *compile-files* and *load-files*, loads the muLISP compiler, compiles the functions defined in *compile-files*, and deletes the compiler. As each function is compiled, COMPILE-FILE displays the function's name, its size in bytes, and its starting address in the muLISP code segment. COMPILE-FILE returns the number of bytes of free space remaining after the compilation.

Note that the files in *load-files* are loaded but *not* compiled. These files should contain function definitions for any external functions called by the functions to be compiled so that the compiler can determine the type of the external functions (LAMBDA, NLAMBDA, or MACRO).

If an error is detected while compiling a file, an error break will occur (see Section 8.4). Selecting the Continue option will continue the file compile beginning with the *next* function. Selecting the Abort option will abort compilation of the file and COMPILE-FILE will immediately return NIL.

```
$ (COMPILE-FILE 'GRAPHICS)          ;Compile GRAPHICS.LSP
Loading: B:GRAPHICS.LSP
Loading: B:COMPILE0.LSP
Loading: B:COMPILE1.LSP
Loading: B:COMPILE2.LSP
Loading: B:COMPILE3.LSP
Compiling: B:GRAPHICS.LSP
SIN-DEG                (244 37457)          ;Compilation data
COS-DEG                (146 37713)
   .                       .
   .                       .
   .                       .
Deleting compiler
320117

$ (SAVE 'GR)                        ;Save compiled functions
```

8.2 BASIC COMPILATION PROCEDURE

This section illustrates the basic procedure for using the muLISP compiler. It also shows the effect of compilation on the performance of some simple benchmark functions. You can do the examples on your computer exactly as they are shown here.

First bring up muLISP, but do *not* yet load the compiler. Now define the functions APP and REV as follows:

```
$ (DEFUN APP (LST1 LST2)                          ;Define APP
    ((NULL LST1) LST2)
    (APP (REV (CDR (REV LST1)))
         (CONS (CAR (REV LST1)) LST2)) )
APP

$ (DEFUN REV (LST)                                ;Define REV
    ((NULL LST) NIL)
    (APP (REV (CDR LST)) (LIST (CAR LST))) )
REV

$ (REV NIL)                                       ;Test REV and APP
NIL

$ (REV '(1))
(1)

$ (REV '(1 2 3))
(3 2 1)
```

APP and REV perform the same computation as the primitive muLISP functions APPEND and REVERSE, respectively. However, APP and REV perform their computations using a deliberately inefficient algorithm. As such, they provide good benchmarks of some basic elements of symbolic computation in LISP: list structure manipulation (CAR, CDR, CONS, and LIST), recursive function calls, and conditional branching.

Given a form to evaluate, the no-eval function TIMING-TEST displays the time it takes to evaluate the form before returning the result of the form evaluation. Define TIMING-TEST as follows:

```
$ (DEFUN TIMING-TEST (NLAMBDA (FORM
      TIME )
    (RECLAIM)
    (TIME T)
    (SETQ FORM (EVAL FORM)
          TIME (TIME))
    (WRITE-STRING (PACK*
      (TRUNCATE TIME 100) \. (REM TIME 100)
      " seconds:  "))
    FORM ))
TIMING-TEST
```

Before compiling REV and APP, establish a timing baseline by running some simple timing tests on these functions:

```
$ (TIMING-TEST (REV '(1 2 3 4 5 6 7)))          ;Series 1
10.05 seconds:  (7 6 5 4 3 2 1)

(TIMING-TEST (REV '(1 2 3 4 5 6 7 8)))
40.21 seconds:  (8 7 6 5 4 3 2 1)
```

(Note: These timing tests were run on an IBM PC with 512K bytes of memory. Different hardware will produce different timing results.)

Now load the muLISP compiler as an incremental compiler as described on page 8-2. To see the effects of compiling, first compile and test REV by itself as follows:

```
$ (COMPILE 'REV)                                ;Compile REV
(REV 100 32401)

$ (TIMING-TEST (REV '(1 2 3 4 5 6 7)))          ;Series 2
5.72 seconds:  (7 6 5 4 3 2 1)
```

```
$ (TIMING-TEST (REV '(1 2 3 4 5 6 7 8)))
22.63 seconds:  (8 7 6 5 4 3 2 1)
```

Compiling REV has improved its execution speed by a factor of approximately 1.8 (= 40.21/22.63) in the two test cases. Of course, REV is still using the *uncompiled* APP. Now compile APP and rerun the timing tests.

```
$ (COMPILE 'APP)                                    ;Compile APP
(APP 164 32529)

$ (TIMING-TEST (REV '(1 2 3 4 5 6 7)))              ;Series 3
3.19 seconds:  (7 6 5 4 3 2 1)

$ (TIMING-TEST (REV '(1 2 3 4 5 6 7 8)))
12.80 seconds:  (8 7 6 5 4 3 2 1)
```

Thus, with *both* APP and REV compiled, there is an overall improvement in execution speed for the two test cases of approximately 3.1 (= 40.21/12.80).

As indicated earlier, compiled functions can be redefined and subsequently recompiled *without* having to change or recompile other functions which called them. To illustrate this, suppose you wished to redefine APP to replace (NULL LST1) with (ATOM LST1). First, make the new definition:

```
$ (DEFUN APP (LST1 LST2)                            ;Redefine APP
    ((ATOM LST1) LST2)
    (APP (REV (CDR (REV LST1)))
         (CONS (CAR (REV LST1)) LST2)) )
APP

$ (TIMING-TEST (REV '(1 2 3 4 5 6 7)))              ;Series 4
5.66 seconds:  (7 6 5 4 3 2 1)

$ (TIMING-TEST (REV '(1 2 3 4 5 6 7 8)))
22.63 seconds:  (8 7 6 5 4 3 2 1)
```

As the increase in run time suggests, the previously compiled REV is now calling the new (and as yet *uncompiled*) definition of APP. REV did not have to be recompiled. Of course, you can now compile the new definition of APP:

```
$ (COMPILE 'APP)                                    ;Recompile APP
(APP 162 32705)

$ (TIMING-TEST (REV '(1 2 3 4 5 6 7)))              ;Series 5
3.24 seconds:  (7 6 5 4 3 2 1)

$ (TIMING-TEST (REV '(1 2 3 4 5 6 7 8)))
12.80 seconds:  (8 7 6 5 4 3 2 1)
```

8.3 COMPILED FUNCTION PERFORMANCE

This section provides an overview of how the compiler improves the execution speed of muLISP programs. Subsections 8.3.1 and 8.3.2 discuss the basic sources of improved execution speed: reducing the interpreter overhead and using in-line machine code for selected muLISP constructions. The net effects of these improvements are then discussed in Subsection 8.3.3.

8.3.1 Reducing Interpreter Overhead

In evaluating an application of a function, the muLISP pseudo-code *interpreter* must work through the expressions in the function's definition analyzing each to determine how it should be evaluated (e.g. as a symbol number, function or special form application, implicit COND, or implicit PROGN), and then evaluating it.

These and other analyses consume time during *each* execution of the function. This time is the overhead paid for the flexibility and convenience of interpretation. The compiler can reduce much of this overhead time.

While compiling a function, the compiler performs similar analyses of the function's definition. Based on these analyses, the compiler generates machine code to replace the function's definition. In effect this "hard-wires" the analyses of the function into machine code -- time need *not* be spent redoing the analyses each time the function is called.

However, since interpretation of the muLISP pseudo-code is highly efficient, the gain in execution speed achieved by compiling functions (see Subsection 8.3.3) is relatively modest.

8.3.2 Using In-line Machine Code

While compiling a function, the compiler translates each form in the function's definition into machine code that computes and returns the value of that form. This generally involves translating function calls of the form ($F\ x_1 \ldots x_n$) into machine code that: a) evaluates and sets up the arguments $x_1, \ldots, x_n$ in accordance with the standard muLISP argument passing conventions (see Chapter 9); and b) passes control to the function *F* that is being applied.

However, the compiler translates certain muLISP constructions directly into in-line machine code. Implicit CONDs and implicit PROGNs are handled with in-line code, as are the following muLISP primitive functions:

Selector Functions: CAR, CDR, CAAR, ..., CDDDDR

Recognizer Functions: SYMBOLP, ATOM, CONSP, NULL

Comparator Functions: EQ, NEQ

Logical Functions: NOT, AND, OR

Assignment Functions: SETQ, PSETQ, POP, PUSH

Control Constructs: QUOTE, PROGN, LOOP, PROG1, IDENTITY, IF, COND, COMMENT, RETURN

In-line code executes much more rapidly than code that sets up arguments and calls a muLISP primitive function. Thus, in writing muLISP functions which you intend to compile, you should use the in-line functions listed above as much as possible.

8.3.3 Net Execution Speed Improvements

By reducing interpreter overhead and using in-line code, the compiler can speed up muLISP functions. However, the compiler *cannot* speed up a) primitive muLISP functions for which there is no in-line code (e.g. numerical functions), b) the garbage collector, and c) input/output (I/O) time. Thus, the net execution speed improvement of compiling a muLISP function will depend on the mix of particular muLISP functions and operations the function uses.

To clarify this, consider the following muLISP definitions for recursive and iterative versions of the factorial function:

```
(DEFUN FACT-R (N)
  ((ZEROP N) 1)
  (* N (FACT-R (SUB1 N))) )

(DEFUN FACT-I (LAMBDA (N I)
  (SETQ I 1)
  (LOOP
    ((EQ N 0) I)
    (SETQ I (* N I))
    (SETQ N (SUB1 N)) ) )
```

This table shows the relative execution speed performance for various calls to these two functions, both uncompiled and compiled.

Function Call	Uncompiled Time (ms)	Compiled Time (ms)	Speed-Up Factor
(FACT-R 0)	0.59	0.31	1.9
(FACT-R 1	1.69	0.83	2.0
(FACT-R 1	14.0	7.7	1.8
(FACT-R 1	215.0	161.0	1.3
(FACT-I 0)	0.99	0.33	3.0
(FACT-I 1)	2.21	0.79	2.8
(FACT-I 1(	15.8	7.7	2.1
(FACT-I 1(	245.0	170.0	1.4

Note that the speed-up factor *decreases* as FACT-R and FACT-I are applied to larger numbers. This is due to the increasing amounts of time required to multiply numbers with the primitive muLISP function *, especially as the products become large. The compiler speed-up effects are gradually diluted by the increases in multiplication time.

Note also that the *iterative* version is improved more by compilation than the *recursive* version (at least for small numbers). This is because the iterative version uses more muLISP primitives which compile to in-line machine code (i.e. SETQ, LOOP, and EQ).

8.4 ERRORS IN COMPILING FUNCTIONS

Errors that occur during the compilation of a function are signaled by an error message that indicates the type of error and the function or form being compiled that caused the error. All errors cause the compilation to stop *without* changing the definition of the function being compiled. The two types of errors that can occur are compilation errors and internal errors.

8.4.1 Compilation Errors

Compilation errors can occur during the translation of a function's definition into assembly language. Most compilation errors are caused by problems which would cause error breaks when encountered by the muLISP interpreter. Thus, a good way to avoid compilation errors is to test and debug functions *before* compiling them. The following paragraphs describe the compilation errors which may occur, showing the form of the error message as it is displayed on the console screen, and describing likely causes.

Non-eval Function Break: *name* indicates that *name* is not a user defined eval function (i.e. not a LAMBDA function) and hence can *not* be compiled. If *name* is a primitively defined function, a special form, an already compiled function, or a user-defined no eval or macro function, it cannot be compiled.

Nonsymbolic Argument Break: *form* indicates that the compiler has encountered *form* which contains a nonsymbolic expression where a symbol is required. Typical forms are (SETQ ...), (PSETQ ...), (PUSH ...), (POP ...), (INCQ ...), and so forth.

Undefined Function Call Break: *form* indicates that the compiler attempted to translate *form* as a function application, but cannot find a definition for the car of *form*. If the car of *form* is a symbol, that symbol must be defined before compiling the function containing *form*.

Insufficient Code Space Break: *n* indicates that there is not enough room left in the muLISP code segment for the *n* bytes of machine code produced in compiling a function. The code segment is *not* garbage collected, so if you have redefined and recompiled functions several times, it may be filled with useless machine code. In that case, you can start a fresh muLISP, load your program, load the compiler, and start over. If, however, your program is just too big to fit into the space available for compiled code, you may still wish to compile selected functions that you identify as being speed critical.

8.4.2 Internal Errors

Internal errors can occur at any point in the compilation process. They indicate some unforeseen problem with the muLISP compiler itself, *not* with the function you are compiling. Internal errors are displayed on the console screen in the following form:

Internal Error #*iiii:j* Break:
$expr_1$
...
$expr_k$

where *iiii:j* is the internal error number, and $expr_1$ through $expr_k$ characterize this particular occurrence of the error.

There is nothing you can do to fix an internal error yourself. You can, however, help to get it fixed by reporting it to Soft Warehouse, Inc., using the muLISP-87 Problem Report form provided at the back of this manual. Please include the entire internal error message, and the definition of the function which caused it.

8.5 POTENTIAL PROBLEM AREAS

This section describes some potential problem areas in using th compiler, and presents means to minimize their effects.

8.5.1 Compiled Function Behavior Changes

A perfect compiler would transform a muLISP function definitio into machine code which would run faster but would otherwise do *exactl* the same things as the original definition.

The muLISP compiler is not quite perfect -- compiling a functio not only speeds it up, but also changes its behavior in other ways. A described in Subsection 8.5.3, functions will *not* respond to consol interrupts after compilation (unless you explicitly insert console interru checks into the function's definition). Moreover, as described i Subsection 8.5.4, functions will exhibit different trace and break behavio after compilation. These behavior changes are necessary for the compile to achieve greater improvements in execution speed.

8.5.2 Compiled Function Compatibility

A compatible compiler would transform a muLISP functio definition into machine code which would run faster but would return th same values and produce the same side-effects as the original definition.

The muLISP compiler is compatible -- compiling a function shoul *not* change its basic computational behavior. Be aware, however, th undiscovered bugs in the compiler could cause it to compile a muLIS function incompatibly. That is, for some arguments, the compiled versio of the function could fail to return, could return different values, c could produce different side-effects than the uncompiled version. Th *functions should always be thoroughly tested after compilation* to mal sure that they still behave properly.

If you encounter a case where a function behaves differently aft compilation, please report it to Soft Warehouse, Inc., using the muLISP-8 Problem Report form provided at the back of this manual. Please inclu

the definition of the function, and describe the difference in behavior before and after compilation.

8.5.3 Console Interrupt Behavior

To improve execution speed, the compiler does not automatically insert console interrupt checks into compiled functions. Thus, the execution of a compiled function cannot generally be stopped by a console interrupt. If you need the interrupt capability in a compiled function, you can direct the compiler to insert a console interrupt check by including the expression

```
(COMMENT CHECK-CONSOLE)
```

in the function's definition at an appropriate point (e.g. at the end of a LOOP construct).

This is *not* an efficient interrupt mechanism. Placing a console interrupt check in a tight loop may slow down a compiled function to an unacceptable level. It may actually be faster and/or more convenient to leave uncompiled any functions that must be capable of being interrupted.

8.5.4 Trace/Break Behavior

At compile time, it is best if all functions to be compiled, as well as the functions called directly in their definitions, are CLEARed of any debugging traces or breaks. Compiling a traced or break-pointed function will simply compile the debug definition, not the function itself. Moreover, if a function being compiled calls a traced or break-pointed primitive function, there will be a smaller improvement in execution speed. This is because the compiler will *not* recognize the traced or break-pointed function as a primitive, and will therefore compile calls to it *indirectly* through FUNCALL or APPLY.

Once compiled, a function can be successfully traced or break-pointed using the muLISP Debugger. Also, traced or break-pointed *user-defined* functions that the compiled function calls will cause a trace or break. However, traced or break-pointed primitive functions and special [illegible] that the compiled function calls will *not* cause a trace or break.

Moreover, a traced or break-pointed compiled function will *not* cause a trace or break on a recursive call to itself. These debugging behavior changes are caused by the fact that a compiled function calls muLISP primitives and itself directly at the machine code level.

Note that these complexities arising from the interaction of the debugger and the compiler are easily avoided by *not* compiling functions until after they have been thoroughly tested and debugged.

8.5.5 Name Conflicts

Besides the functions COMPILE, DELETE-COMPILER and COMPILE-FILE, the compiler uses and generates a variety of other symbols. If your program redefines sets, or otherwise uses any compiler symbols, the resulting name conflicts may cause unpredictable results both for the compiler and for your program.

To help avoid name conflicts, compiler symbols have been given distinctive patterns:

1. Except for COMPILE, DELETE-COMPILER, and COMPILE-FILE, all compiler functions, global and special variables, constant property indicators, and CATCH tags end with !C.

2. All constant symbols generated by the compiler to refer to numbers and quoted lists will have the form K!C*n*, for some integer *n*.

3. All assembly language labels generated during compilation will have the form !C*n*, for some integer *n*.

Thus, to minimize the chance of name conflicts, simply avoid using the string !C in the function names, property indicators, and other symbols you use. Also, the following symbols should not be used as the names of functions to be compiled, since the compiler uses them for the names of 8086 registers and jump conditionals:

AX	BX	CX	DX	SI	DI	BP	SP	CS	DS	IS	SS
E	B	G	NE	NB	GE	A	C	L	NA	NC	LE

8.5.6 Redefining Primitive muLISP Functions

The compiler is designed to operate with the standard primitive muLISP functions. Redefining primitive functions can cause the following problems:

1. Redefining primitive functions used by the compiler may cause it to malfunction; such malfunctions may *not* be detected by the compiler's internal error checking.

2. To enable incremental modification, compiled functions call user-defined functions *indirectly* through FUNCALL or APPLY. Therefore, redefining FUNCALL or APPLY may cause incorrect linkage from compiled code to other user-defined functions.

3. To achieve maximum execution speed, compiled functions call primitive functions *directly* rather than going through FUNCALL or APPLY. Therefore, if a function that calls a primitive is compiled *before* the primitive is redefined, the compiled function will continue to call the original definition of the primitive.

4. Redefining primitive functions which the compiler translates directly into in-line machine code will *not* have the intended effect: the compiler will generate the normal in-line code for these primitives no matter what their definition.

You can avoid such problems by using the standard muLISP primitive definitions in programs you wish to compile. If, however, you must redefine some primitive functions -- e.g. the undefined function error handler or the muLISP driver -- complete *all* redefinitions *before* compiling any functions. This should minimize the potential for inconsistencies since the compiler recognizes primitive functions as symbols which have a numeric definition falling within the primitive muLISP code space. Thus, if you move a primitive definition to a new symbol, the compiler will treat the new symbol as a primitive function, compiling calls to it directly. Moreover, if you redefine a primitive symbol with muLISP code, the compiler will treat the primitive as a user-defined function, compiling calls to it indirectly.

9. MACHINE LANGUAGE ROUTINES

This chapter is for the small number of muLISP users who may find it necessary to write machine language routines and call them from muLISP. There are three major reasons for writing such routines: a) to control some computer hardware or peripheral device; b) to communicate with the host operating system; c) to implement some critical user-defined functions in machine language for efficiency reasons. The services provided by the Hardware Interface Functions (see Chapter 5) are intended to minimize as much as possible the need to write machine language routines.

This chapter describes how to write and link to machine language routines if you should need to do so. It assumes that you are thoroughly familiar with the muLISP data structures as described in Chapter 4 of this manual and with the architecture and assembly language of the 8086 microprocessor.

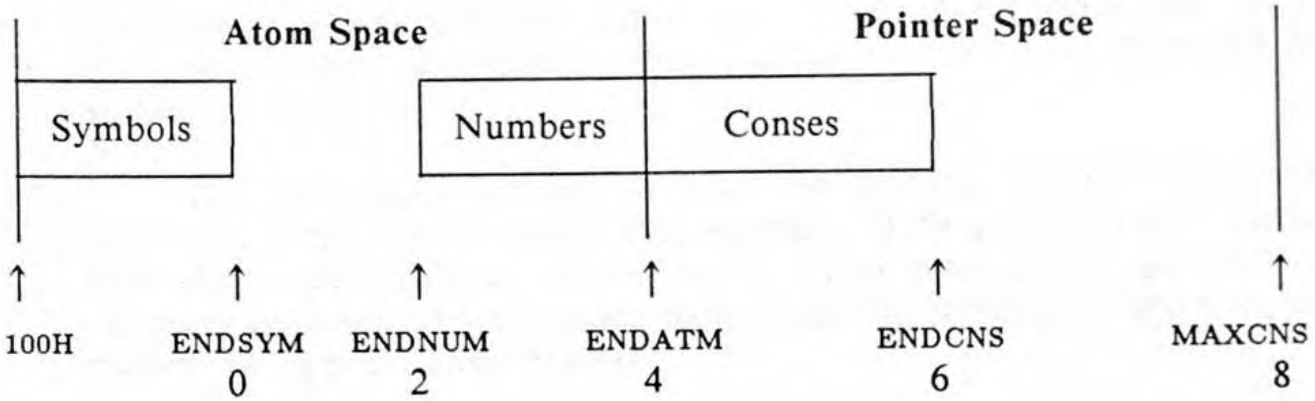
Appendix B to this manual is an assembly language listing of the muLISP **base page**. The base page consists of the first 300H bytes of MULISP.COM. It contains the default values for several system variables (e.g. computer type, line length, radix base, etc.), the default readtable, the macro character function table, the ten pseudo-registers, the line editor command table, and space for entry points into user-defined special form routines.

The commented listings show the addresses of these items in the event you need to change their default value. The change can be done and a revised MULISP.COM can be saved using the assembly language debug program supplied with your computer's operating system.

Appendix C to this manual is an assembly listing of the muLISP **global storage area**. This area contains the variables (i.e. two byte pointers) that delimit the ends of the various muLISP data spaces and other system variables. These variables can be accessed from assembly language programs beginning at an offset of 0 in the data segment.

9.1 DATA REPRESENTATION

In order to write machine language routines, you must have an understanding of how and where muLISP stores data objects in memory. The following diagram illustrates where the three types of objects are stored:



The diagram also gives the names of the variables that point to the current end of the various regions. These pointers are maintained by muLISP in the global storage area at the offsets indicated under their name. As an efficiency measure, the register DX contains the same value as ENDATM.

Using these variables, it is easy for a machine language routine to determine the type of a data object (a symbol, a number, or a cons) merely from its address. For example, if SI points to the object, the following code jumps to one of three routines based upon the object's type:

```
ENDSYM  EQU     0
ENDNUM  EQU     2
ENDATM  EQU     4

        CMP     SI,WORD PTR DS:[ENDSYM]
        JB      LABEL1          ;Jump if SI is a symbol
        CMP     SI,DX           ;Note that DX = ENDATM
        JB      LABEL2          ;Jump if SI is a number
        JMP     LABEL3          ;SI is a cons
```

Unfortunately the 8086 divides its 20 bit address space into **segments** which can only be a maximum of 64 Kilobytes each. The pointer cells making up a muLISP data object are stored at the same offset in different segments. The object's car cell is stored in the segment pointed to by the DS segment register. Its cdr cell is stored in the segment pointed to by the ES segment register. If the object is an atom it has a third and fourth cell which are stored in the definition and print name pointer segments respectively. The 16 bit paragraph addresses of these segments are stored in the variables DEFSEG and PNPSEG in the global storage area.

Knowing this, it is easy for machine language routines to access the contents of an object's cells. For example, if SI points to a symbol, the following code loads a pointer to its value into AX, a pointer to its property list into BX, a pointer to its function definition into CX, and a pointer to its print name into DI:

```
DEFSEG  EQU     10H
PNPSEG  EQU     12H

        MOV     AX,[SI]         ;AX points to symbol's value
        MOV     BX,ES:[SI]      ;BX points to symbol's plist
        PUSH    DS
        MOV     DS,WORD PTR DS:[DEFSEG]
        MOV     CX,[SI]         ;CX points to symbol's defn
        POP     DS
        PUSH    DS
        MOV     DS,WORD PTR DS:[PNPSEG]
        MOV     DI,[SI]         ;DI points to symbol's pname
        POP     DS
```

Since the symbol's value and property list must be bona fide muLISP objects, they can be accessed as described earlier. If the definition pointer in CX is odd, it contains the offset of a machine language routine in the code segment (CS). If CX is even, it contains the offset of D-code, which is stored in the stack segment (SS). The print name pointer in DI contains the offset of the ASCII print name string which is stored in the print name string segment (PNSSEG).

The contents of the four cells making up a number can be found by the same code as shown above for obtaining a symbol's four cells. The vector cell of a big integer contains the offset of the number vector which is stored in the number vector segment (VECSEG). Note that VECSEG and PNSSEG are variables maintained in the muLISP global storage area.

9.2 ARGUMENT PASSING

User-defined machine language routines can be either **eval functions** or **special forms** (see Evaluation Functions section of Chapter 5). The entry point for eval function routines must be on an *odd* address. The entry point for special forms routines can be on an even or odd address since they are only referenced indirectly via the jump table located in the base page.

Arguments to functions are evaluated, and the address of the resulting object is passed to the machine language routine above the top of the variable (BP) stack. On entry, CX will contain two times the number of actual arguments that were in the call to the function. If the following routine is called at the beginning of a function, it loads SI, DI, and BX with the first three arguments to the function. If there are not enough arguments, it loads the registers with NIL:

```
NIL      EQU     100H          ;Offset of NIL is a constant

GETTHR:  CMP     CX,6
         JB      GETTWO        ;Jump if less than 3 arguments
         MOV     SI,[BP]       ;SI: argument one
         MOV     DI,[BP+2]     ;DI: argument two
         MOV     BX,[BP+4]     ;BX: argument three
         RET

GETTWO:  MOV     BX,NIL        ;BX: NIL
         CMP     CX,4
         JB      GETONE        ;Jump if less than 2 arguments
         MOV     SI,[BP]       ;SI: argument one
         MOV     DI,[BP+2]     ;DI: argument two
         RET

GETONE:  MOV     DI,NIL        ;DI: NIL
         JCXZ    GETZRO        ;Jump if no arguments
         MOV     SI,[BP]       ;SI: argument one
         RET

GETZRO:  MOV     SI,NIL        ;SI: NIL
         RET
```

Since the arguments to special forms are not evaluated, the address of the actual argument list is passed to the routine in the SI register. Since the special form could have been invoked either from a compiled function or from evaluating an S-expression, the list can be either D-code or a linked list. You must write separate routines to handle each case. The offset address of the two routines is then stored in a two entry jump table. Space for 10 special form jump tables has been set aside in the base page for user-defined special forms. The address of the D-code handler is stored in the first entry; the address of the S-expression handler in the second.

The value returned by a machine language routine is the object pointed to by the DI register. DI must contain the offset address of a bona fide muLISP data object. If the routine does not need to return a value, DI should still be set to the offset of NIL which is 100H.

9.3 CALLING PRIMITIVE ROUTINES

User-defined routines can call the large number of primitively defined muLISP function and special form routines. These primitive routines use the same **calling convention** to receive arguments and return values as that described for user-defined routines. The offset address in the code segment of the functions can be determined from muLISP using the function GETD:

```
$ (SETQ *PRINT-BASE* 16)                    ;Print in hexadecimal
10

$ (GETD 'CONS)                              ;Get CONS's entry point
0B27

$ (GETD 'LOOP)                              ;Get LOOP's jump table
1B8

$ (SETQ *PRINT-BASE* 10)                    ;Print in decimal
10
```

Note that user-defined routines must save the DX and the four 8086 segment registers, so that they can be restored to their original value before a call is made to a primitive routine and before control is returned to muLISP.

Calls to muLISP routines that do consing or generate numbers can invoke the garbage collector. Garbage collections will destroy unreferenced muLISP data objects and can change the address of referenced objects. To ensure that pointers to objects are saved and updated during a garbage collection, they must be pushed onto the muLISP variable stack. The BP register is the variable stack pointer. The following code segment illustrates how the pointer in BX is pushed on and later popped off the variable (BP) stack for a call to the muLISP CONS routine:

```
CONS    EQU     0B27H

        MOV     [BP],BX         ;Push BX on BP stack
        INC     BP              ;Increment BP stack pointer
        INC     BP
        MOV     AX,OFFSET CONS+3
        EVEN                    ;Force CALL to an odd address
        NOP
        CALL    AX              ;Set DI to CONS (SI, DI)
        DEC     BP              ;Decrement BP stack pointer
        DEC     BP
        MOV     BX,[BP]         ;Pop updated BX from BP stack
```

Note the use of the EVEN and NOP opcodes immediately preceding the call to CONS. The assembly language instruction CALL AX is a two byte instruction. Therefore, if the call instruction begins on an odd address, the return address that is pushed on the SP stack will be odd.

During a reallocating garbage collection references on the SP stack to D-code must be updated when the D-code is moved during reallocation. To distinguish D-code pointers from return addresses on the SP stack, the reallocator relies on the fact that D-code pointers are even whereas return addresses are odd. Therefore, it is necessary to ensure that a return address (or anything else for that matter) that is pushed on the SP stack is odd *if* a garbage collection can occur while the return address is on the stack.

9.4 LOADING AND LINKING

A three step process is used to load and link user-defined machine language routines into muLISP. First, the function ALLOCATE is called to set aside enough space in the code segment for the routine. Next, the function BINARY-LOAD is called to actually load the routine into memory. Finally, the function PUTD is called to link a symbol's function definition cell to the routine.

If *bytes* is a positive integer, (ALLOCATE *bytes*) frees *n* bytes of memory in the code segment and returns the offset of the base of the newly allocated memory. Note that if ALLOCATE returns an integer, it will be an even integer.

If *offset* is a positive integer less than 65536 and the file on the drive and directory specified by *filename* exists, (BINARY-LOAD *filename offset*) loads *filename* into the muLISP code segment at *offset* and returns the number of bytes loaded. If no file name extension is specified in *filename*, BINARY-LOAD assumes a file name extension of BIN. Normally the *offset* in a call to BINARY-LOAD is the offset address returned by the call to ALLOCATE plus 1 to make it odd.

If *name* is a symbol and *n* is the odd entry address (i.e. its offset in the code segment) of a user-defined machine language function routine, (PUTD *name n*) links *name* to the function. If *n* is the even offset address of a special form jump table in the base page, (PUTD *name n*) links *name* to the special form routines pointed to by the table.

The following function (LOAD-LINK *name filename bytes*) is a useful utility that automates the process for loading and linking user-defined machine language routines:

```
(DEFUN LOAD-LINK (NAME FILE-NAME BYTES
    OFFSET )
  ((SETQ OFFSET (ALLOCATE BYTES))       ;Allocate memory
    ((BINARY-LOAD FILE-NAME (ADD1 OFFSET)) ;Load file
      (PUTD NAME (ADD1 OFFSET)) ) ) )    ;Link function
```

As an example of the major points discussed in this chapter, consider the problem of converting the following user-defined function into a machine language routine:

```
(DEFUN COPY (OBJ)
  ((ATOM OBJ) OBJ)
  (CONS (COPY (CAR OBJ)) (COPY (CDR OBJ))) )
```

COPY is equivalent to the primitive function COPY-TREE. Appendix D to this manual is a listing of the equivalent 8086 assembly language routine for COPY. If the routine is assembled and turned into the file COPY.BIN, it can be loaded and linked to the symbol COPY as follows:

```
$ (LOAD-LINK COPY B:COPY 60)          ;Load and link COPY
COPY

$ (COPY '(A (B . C) D))               ;Call the new function COPY
(A (B . C) D)
```

10. LISP AND AI REFERENCES

After learning muLISP, you may be anxious to learn more about AI (artificial intelligence) programming techniques. There is an extensive body of literature on AI and LISP programming. This chapter lists some of the AI professional societies, periodicals, and books. They can serve as an entry point to this exciting field.

10.1 PROFESSIONAL SOCIETIES

The American Association for Artificial Intelligence (AAAI), 445 Burgess Drive, Menlo Park, CA, 94025; phone (415) 328-3123.

The ACM Special Interest Group on Artificial Intelligence (SIGART), Association for Computing Machinery, 1133 Avenue of the Americas, New York, NY, 10036.

10.2 PERIODICALS

AI Expert, Miller Freeman Publications, 500 Howard St., San Francisco, CA, 94105; phone (415) 397-1881.

AI Magazine, American Association for Artificial Intelligence, 445 Burgess Drive, Menlo Park, CA, 94025; phone (415) 328-3123.

AI Today, 104 Frame Road, Elkview, WV, 25071; phone (304) 965-5548.

AIWeek, Applied Artificial Intelligence Reporter, University of Miami Intelligent Computer Systems Research Institute, P.O. Box 248235, 421C Jenkins Building, Coral Gables, FL, 33124; phone (305) 284-5161.

Artificial Intelligence, P.O. Box 211, 1000 AE Amsterdam, The Netherlands.

Expert Systems, Learned Information (Europe) Ltd., Woodside, Hinksey Hill, Oxford OX1 5AU, England; phone Oxford (0865) 730275; telex 837704 INFORM G. *or* Learned Information Inc., 143 Old Marlton Pike, Medford, NJ, 08055, USA; phone (609) 654-6266.

Expert Systems User, Expert Systems User Ltd., 52-54 Southwark Street, London SE1 1UN, Great Britian; phone 01-407 7400; telex 9413548.

IEEE Expert, IEEE Computer Society, 345 East 47th St., New York, NY, 10017.

Machine Intelligence, Ellis Horwood Ltd., Market Cross House, Cooper St., Chichester, West Sussex, PO19 1EB, England.

PC AI, Knowledge Technology, Inc., 3310 West Bell Rd., Ste. 119, Phoenix, AZ, 85023; phone (602) 439-3253.

Pattern Recognition, Pergamon Press Inc., Maxwell House, Fairview Park, Elmsford, NY, 10523.

10.3 BOOKS

Allen, J. R., Anatomy of LISP, McGraw-Hill Book Company, New York NY, 1978.

Anderson, J. R., Corbett, A. T., and Reiser, B. J., Essential LISP Addison-Wesley Publishing Company, Inc., Reading, MA, 1987.

Barr, A., and Feigenbaum, E. A., (eds), The Handbook of Artificial Intelligence, William Kaufmann, Inc., Los Altos, CA, 1982.

Berk, A. A., LISP, The Language of Artificial Intelligence, Van Nostrand Reinhold Company, New York, NY, 1985.

Berkeley, E. C., and Bobrow, D. G., (eds), The Programming Language LISP: Its Operation and Applications, The M.I.T. Press, Cambridge MA, 1964.

Bromley, H., LISP Lore: A Guide to Programming the LISP Machine Kluwer Academic Publishers, Hingham, MA, 1986.

Brookes, R. A., Programming in Common LISP, John Wiley & Sons, New York, NY, 1985.

BYTE: The Small Systems Journal, LISP Issue, Vol 4, No 8, BYTE Publications Inc., Peterborough, NH, August 1979.

Charniak, E., et al, Artificial Intelligence Programming, Lawrence Erlbaum Associates, 365 Broadway, Hillsdale, NJ, 1980.

Friedman, D. P., The Little LISPer, Science Research Associates Inc. London, England, 1974.

Gabriel, R. P., Performance and Evaluation of LISP Systems, The M.I.T Press, Cambridge, MA, 1985.

Hasemer, T., Looking at LISP, Addison-Wesley Publishing Co., Reading MA, 1984.

Henderson, P., Functional Programming: Application and Implementation Prentice-Hall, Englewood Cliffs, NJ, 1980.

McCarthy, J., Recursive Functions of Symbolic Expressions and Their Computation by Machine, Communications of the ACM, pages 184-195, April 1960.

McCarthy, J., et al., LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge, MA, 1963.

Maurer, W. D., A Programmer's Introduction to LISP, American Elsevier, New York, NY, 1973.

Queinnec, C., LISP, John Wiley & Sons, New York, NY, 1983.

Rich, E., Artificial Intelligence, McGraw-Hill Book Company, New York, NY, 1983.

Schoffa, G., Die Programmiersprache LISP, Franzis-Verlag GmbH, Munchen, Germany, 1987.

Siklossy, L., Let's Talk LISP, Prentice-Hall, Englewood Cliffs, NJ, 1976.

Steele, G. L. Jr., Common LISP: The Language, Digital Press, Burlington, MA, 1984.

Touretzky, D. S., LISP: A Gentle Introduction to Symbolic Computation, Harper and Row, New York, NY, 1983.

Weinreb, D. L. & Moon, D., LISP Machine Manual, Artificial Intelligence Laboratory, MIT, Cambridge, MA, 1981.

Weissman, C., LISP 1.5 Primer, Dickenson Publishing Co., Belmont, CA, 1968.

Wilensky, R., Common LISPcraft, W. W. Norton & Company, New York, NY, 1986.

Winston, P. H. & Horn, B. K. P., LISP, Second Edition, Addison-Wesley Publishing Co., Reading, MA, 1984.

11. GLOSSARY

Address typing: a method of determining the type of a data object (e.g. name, number, list, etc.) by its address in memory. This method of data typing does not require the use of tag bits, nor is there a need to actually examine the object merely to determine its type.

Algorithm: a set of rules for solving a problem in a finite number of steps.

Argument: an expression whose value is used by a function to compute a result. When a function is called, the arguments are evaluated and the values are passed to the function.

Artificial Intelligence (AI): a field of computer science that attempts to make computers perform tasks that are normally done by humans such as reasoning, learning, and creating. This can further our understanding of intelligence itself.

ASCII (American Standard Code for Information Interchange): this code is the widely used standard for transmitting and storing character data by computers. The letters of the alphabet, decimal digits, punctuation marks, and other special characters are each represented by a unique 7 bit code.

Assembly language: the low-level computer language for a processor each statement of which corresponds one-for-one to a machine language instruction of the processor. Assembly language is used to write muLISP since maximum speed and compactness can be achieved through its use.

Binary: a condition in which there are but two alternatives. Thus the binary number system (i.e. base 2) has but two values for its digits either 0 or 1.

Bit: a binary digit, the smallest unit of computer data. It can be either 0 or 1. However, collections of bits can represent arbitrarily large and complex numbers and data.

Boolean logic: a binary system of logic named for George Boole. It includes operators such as AND, OR, and NOT which return truth values based upon the truth values of their operands. Thus boolean operators are widely used in programming to combine logical values to determine the desired course of action.

Bootstrap loader: a program usually in ROM which is designed to load the operating system and then turn control over to the freshly loaded system. Initiating the loader on a computer may either be automatic when the system is turned on or reset, or it may be necessary to perform a number of actions manually.

Byte: a unit of data composed of 8 bits. The meaning of the information stored depends entirely on the way in which the byte is interpreted. For instance, a byte can represent an ASCII character or an integer in the range from 0 to 255.

Compiler: a program that converts programs written in a higher level computer language into a lower-level form that is more easily "understood" by the computer. Compilers can produce either directly executable machine code, or easily interpreted and portable pseudo-code. Compilers do not actually execute programs. Program execution is the job of the microprocessor or, in the case of pseudo-code, the interpreter.

Control character: a nonprintable ASCII character code used primarily to communicate control directives to the receiving device. Examples include the carriage return, linefeed, and bell characters.

CRT (Cathode Ray Tube): a computer console screen that uses an electronic vacuum tube to display characters and/or graphics.

Cursor: a rectangular block or horizontal bar of light, often blinking, on a CRT screen, whose position indicates where the next character will be displayed. The cursor position is controlled by special control characters sent from the computer to the terminal.

Disk drive: a computer accessory device that rotates disks (hard or floppy) and reads and writes information on the disk by means of a magnetic head sensor.

Environment: the current state of a program in the computer. For muLISP programs it includes the values and properties of all program variables and the definitions of all functions.

File: A block of data stored on the mass storage device under a given name. It consists of a collection of records treated as units.

Floppy disks: flexible plastic disks coated with an iron oxide film that are capable of storing magnetically recorded information. They are used in conjunction with floppy disk drives as a mass storage device.

Function: a procedure for computing from a given set of arguments a unique result. In most programming languages, functions can have side effects which permanently affect the environment.

Global variable: a variable whose value has not been assigned by its use in the formal argument list of a function. Any change to such a variable is recognized by all functions which use it.

Hard disks: rigid disks covered with a magnetic oxide film capable of storing large amounts of data (tens of millions of bytes). Since the disks rotate at a high rate of speed, hard disk drives can access data quickly. Currently hard disk systems are more expensive and hence less common than floppy disk storage systems.

Hardware, computer: the physical components making up a computer system. It includes the computer's power supply, electronic logic circuits, and the frame housing the computer.

Head alignment: in order for a disk drive to read or write information from or to a given track of a floppy disk, the magnetic head sensor must be accurately positioned over the track. If this is not the case and disk errors are occurring, an alignment must be performed.

Hexadecimal: a notation for representing numbers using base 16 rather than base 10. The successive hexadecimal digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Since 16 is a power of 2, it easy to convert from the binary representation used by computers to the hexadecimal notation more easily understood by humans.

Interpreter: a computer program that implements a computer language by executing while decoding individual program statements written in that language.

I/O (Input/Output): refers to the transfer of data into and out of a device such as a disk drive or a computer.

Kilobyte: 1024 bytes of storage. This number is used as the standard unit for measuring computer memory sizes since it is an integer power of 2.

Local variable: a variable listed in a function's formal argument list. A function's local variables can be modified within that function without having any effect on identically named variables that occur outside the function. This is a tremendous advantage since it frees the programmer from concerns about undesirable side-effects.

Logged drive: the disk drive which will be used by default in the event a specific drive is not specified in a command related to disk file I/O. The currently logged drive can be changed by an operating system command.

Lower-case: the letters of the alphabet expressed as noncapitals.

Machine code: instructions that are directly executable by a microprocessor.

Mass storage device: hardware designed to provide permanent storage of computer information, such as floppy or hard disk drives. Although not as fast as RAM, it is usually capable of storing large amounts of information, and for the most part, the data is immune to computer malfunctions.

Megahertz (MHz): a unit of frequency equal to a million cycles per second.

Microprocessor: the electronic device which actually performs the logical and arithmetic operations in a microcomputer. It consists of a small silicon chip housed in a ceramic or plastic dual in-line package (DIP) usually having forty or more connector pins.

Monitor: a small supervisor program, usually implemented in PROM. It provides a primitive means of communicating with the computer to allow such operations as examining and changing memory.

Nonvolatile memory: computer memory which retains its information even after the computer has been turned off. Thus it provides safer, more permanent storage than volatile memory such as RAM.

On-line: information which can be directly read by a computer. Thus data on a mass storage device is on-line whereas printed matter is off-line.

Operating System: a complete system which provides the basis for user interaction with a computer. It usually provides facilities for the generation, testing, and running of programs and the capability for permanently storing such programs for future use.

Parse: to determine the syntactic relationships between the components of a statement in a given computer language.

Pseudo-code: a code for storing function definitions that is extremely compact and can be efficiently interpreted by a computer. Pseudo-code has the additional advantage over machine code of being "portable" since it is relatively easy to program different microprocessors to interpret the same pseudo-code.

RAM (Random Access Memory): the primary means of fast-access read/write data storage within microcomputers. Data stored in RAM is accessed by specifying the address of the data. In contrast to sequential access memory, the time for such an access is independent of the location of the data.

Rational arithmetic: has for its domain or area of applicability all numbers which can be expressed as the ratio of two integers. Thus it includes integer arithmetic as a subset.

Record: a block of data from a disk file which is stored as a unit on a mass storage device.

Register: a storage cell located in a microprocessor that is used by the processor for logical and arithmetic operations. The capacity of a register is measured by the number of bits it can store. Common microprocessor register sizes include 8 bits and 16 bits.

Reset: a procedure to return a computer to the initial condition it was in when first turned on. If a computer program goes wild, reset the computer to initiate the bootstrap loader process. A reset is usually initiated by pressing a switch on the computer or a combination of keys on the keyboard.

ROM (Read Only Memory): this form of data storage can only be read from, but not changed, by the computer. However, it has an advantage over most RAM's in that it is a nonvolatile memory. Thus, it is commonly used to store bootstrap loader programs. ROM is random access memory; however, it is not called RAM since that term is reserved for the volatile, read-write memory.

Sector: a wedge shaped area on the surface of a floppy or hard disk. That portion of a track which is contained in one sector is capable of storing one record of data.

Software, computer: the program or set of instructions which specify how a computer is to accomplish a given task. This is considered "soft" since the same computer hardware can execute many different programs.

Stack, push-down: a data structure used to temporarily store information. The PUSH operation stores data on top of a stack pushing older data "down". The POP operation retrieves the most recently pushed data from the top of a stack. Stacks are heavily used in recursive languages such as LISP and muSIMP.

Syntax: the patterns of symbols or tokens that constitute all the valid expressions or statements of a computer language.

Track: one of many concentric circles around a disk in which data is stored. That portion of a track which is contained in one sector is capable of storing one record of data.

Truth value: either the value TRUE or FALSE. Truth values are the subject of Boolean Logic and play a fundamental role in program control.

Upper-case: the letters of the alphabet expressed in capital letters.

Variable: a name which is used by a computer program to store a value. The value can be explicitly made using an assignment statement, or implicitly made when a function call assigns values to the function's local variables.

A. IBM PC EXTENDED FUNCTION KEYS

The IBM PC keyboard has a set of **extended function keys**. They include the ten function keys located on the left or top of the keyboard and the cursor control keys located on the right side of the keyboard. See the IBM PC's Guide to Operations Manual and the BASIC Programming Manual for details.

If you are running muLISP on an IBM PC, programs can detect that the user has pressed one of the extended function keys if READ-BYTE returns a 255. The *next* byte returned by READ-BYTE indicates which extended function key was pressed as shown in this table.

Byte	Key	Byte	Key	Byte	Key	Byte	Key
15	SHIFT |<—	48	ALT-B	85	SHIFT-F2	109	ALT-F6
16	ALT-Q	49	ALT-N	86	SHIFT-F3	110	ALT-F7
17	ALT-W	50	ALT-M	87	SHIFT-F4	111	ALT-F8
18	ALT-E	59	F1	88	SHIFT-F5	112	ALT-F9
19	ALT-R	60	F2	89	SHIFT-F6	113	ALT-F10
20	ALT-T	61	F3	90	SHIFT-F7	114	CTRL-PrtSc
21	ALT-Y	62	F4	91	SHIFT-F8	115	CTRL <—
22	ALT-U	63	F5	92	SHIFT-F9	116	CTRL —>
23	ALT-I	64	F6	93	SHIFT-F10	117	CTRL-End
24	ALT-O	65	F7	94	CTRL-F1	118	CTRL-PgDn
25	ALT-P	66	F8	95	CTRL-F2	119	CTRL-Home
30	ALT-A	67	F9	96	CTRL-F3	120	ALT-1
31	ALT-S	68	F10	97	CTRL-F4	121	ALT-2
32	ALT-D	71	Home	98	CTRL-F5	122	ALT-3
33	ALT-F	72	↑	99	CTRL-F6	123	ALT-4
34	ALT-G	73	PgUp	100	CTRL-F7	124	ALT-5
35	ALT-H	75	<—	101	CTRL-F8	125	ALT-6
36	ALT-J	77	—>	102	CTRL-F9	126	ALT-7
37	ALT-K	79	End	103	CTRL-F10	127	ALT-8
38	ALT-L	80	↓	104	ALT-F1	128	ALT-9
44	ALT-Z	81	PgDn	105	ALT-F2	129	ALT-0
45	ALT-X	82	Ins	106	ALT-F3	130	ALT -
46	ALT-C	83	Del	107	ALT-F4	131	ALT-=
47	ALT-V	84	SHIFT-F1	108	ALT-F5	132	CTRL-PgUp

B. muLISP BASE PAGE

Appendix B is an annotated assembly language listing of the muLISP base page. It shows the initial or default value for many system variables. The first two columns in each row of the listing are the hexadecimal and decimal offsets in the code segment of the variables.

```
                ORG     100H    ;muLISP base address
0100    256     JMP     LISP    ;Jump to entry point

                DS      125     ;Reserved space, do not use
```

The following **Special Form Jump Table** provides space for up to 10 pairs of pointers to user-defined special forms. The first entry of each pair points to D-code; the second points to the S-expr routine. See Chapter 9 for details on implementing user-defined special forms.

```
0180    384     DW      0,0
0184    388     DW      0,0
0188    392     DW      0,0
018C    396     DW      0,0
0190    400     DW      0,0
0194    404     DW      0,0
0198    408     DW      0,0
019C    412     DW      0,0
01A0    416     DW      0,0
01A4    420     DW      0,0

01A8    424     DS      88      ;Reserved space, do not use
```

The following **Default Read Table** specifies the default character type for the 256 different ASCII characters. The bits in each byte in the table describe the type of the corresponding ASCII character as follows:

Bit	Mask	Character type	Bit	Mask	Character type
0	1	Whitespace	4	10H	Interrupt
1	2	Single escape	5	20H	Terminating macro
2	4	Multiple escape	6	40H	Nonterminating macro
3	8	Break	7	80H	Comment macro

```
;                 @ A B C D E F G H I J K L M N O
0200 512   DB   0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0

;                 P Q R S T U V W X Y Z  [ \ ] ^ _
0210 528   DB   0,0,0,0,0,0,0,0,0,0,0,16,0,0,0,0

;                   !  " # $ % &  '  (  ) * +  , - . /
0220 544   DB   1,8,40,8,8,8,8,40,40,40,8,8,40,8,8,8

;                 0 1 2 3 4 5 6 7 8 9 :  ;  < = > ?
0230 560   DB   0,0,0,0,0,0,0,0,0,0,8,128,8,8,8,8

;                 @ A B C D E F G H I J K L M N O
0240 576   DB   8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

;                 P Q R S T U V W X Y Z [  \  ] ^ _
0250 592   DB   0,0,0,0,0,0,0,0,0,0,0,8,10,40,8,8

;                 ' a b c d e f g h i j k l m n o
0260 608   DB   8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

;                 p q r s t u v w x y z {  | } ~ RUB
0270 624   DB   0,0,0,0,0,0,0,0,0,0,0,8,12,8,8,0

0280 640   DB   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

0290 656   DB   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02A0 672   DB   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02B0 688   DB   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02C0 704   DB   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02D0 720   DB   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02E0 736   DB   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02F0 752   DB   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

The following **Default Macro Character Function Table** contains pointers to the default definitions for macro characters. The first word of the table is the number of entries in the table. There is room for up to 10 entries. An entry consists of a byte containing the ASCII code for a character followed by a word pointing to its macro definition; odd pointers imply machine language routines, even imply D-code.

```
0300   768    DW     4              ;Entries in table
0302   770    DB     "'"            ;Macro character
              DW     RDQUOTE        ;Macro function pointer
0305   773    DB     '('
              DW     RDLIST
0308   776    DB     ';'
              DW     RDCOMMENT
030B   779    DB     '"'
              DW     RDSTRING
030E   782    DB     0
              DW     0
0311   785    DB     0
              DW     0
0314   788    DB     0
              DW     0
0317   791    DB     0
              DW     0
031A   794    DB     0
              DW     0
031D   797    DB     0
              DW     0
```

The following **Pseudo-registers** are used by REGISTER & INTERRUPT.

```
0320   800    DW     0      ;0      AX
0322   802    DW     0      ;1      BX
0324   804    DW     0      ;2      CX
0326   806    DW     0      ;3      DX
0328   808    DW     0      ;4      SI
032A   810    DW     0      ;5      DI
032C   812    DW     0      ;6      BP
032E   814    DW     0      ;7      DS
0330   816    DW     0      ;8      ES
0332   818    DW     0      ;9      FLAGS
```

If the following minimum number of bytes are not free after a garbage collection and data space reallocation, a "Memory Full" error break occurs.

```
0334   820    DW     200H    ;Minimum atom space bytes
0336   822    DW     200H    ;Minimum pointer space bytes
0338   824    DW     400H    ;Minimum stack space bytes
```

The data space reallocator allocates stack space so that the total free space to stack free space is the following.

```
033A   826    DW     32      ;Total free to stack free space

033C   828    DW     1       ;Default precision setting (single)
033E   830    DW     0       ;Default underflow setting (disabled)
0340   832    DW     7       ;Default value of *PRINT-POINT*

0342   834    DW     CSREG   ;Initial value of the CS register
```

A data space reallocation occurs after a garbage collection if, for any space, the ratio of the size of the space to the optimum size of the space is greater than [0344H]/[0346H] or less than [0346H]/[0344H].

```
0344   836    DW     4       ;Reallocation sensitivity
0346   838    DW     3       ;Reallocation sensitivity
```

muLISP will *not* use memory above the paragraph address in the following word. If it is zero, muLISP uses all available memory.

```
0348   840    DW     0       ;Limit on muLISP memory size
```

The following word is the default file and printer line length.

```
034A   842    DW     79      ;Default file line length
```

The following bytes are the initial radix base used for I/O.

```
034C   844    DB     10      ;Initial value of *READ-BASE*
034D   845    DB     10      ;Initial value of *PRINT-BASE*
```

The following byte is used as a mask on character input from files to strip parity bit.

```
034E   846    DB     7FH     ;File input character mask
```

The following byte is the character output by muLISP to sound the console bell. Set byte to 0 to deactivate bell.

```
034F   847    DB     7          ;Console bell character
```

The following bytes are the characters used by PRIN1 to delimit special characters when the control variable *PRINT-ESCAPE* is nonNIL.

```
0350   848    DB     '\'        ;Single escape character
0351   849    DB     '|'        ;Multiple escape character
```

The following byte controls the frequency at which the console is checked for escape char. Value can range from 1 for most frequent to 255 for least.

```
0352   850    DB     100        ;Keyboard interrupt sensitivity
```

If the following byte is nonzero, whitespace characters that terminate a READ will be unread so that they can be reread by next READ.

```
0353   851    DB     -1         ;Read preserving white space flag
```

If the following byte is nonzero, an "End-Of-File" error break occurs when the EOF character is encountered while reading a file. If zero, EOF error break only occurs when the physical EOF is reached.

```
0354   852    DB     -1         ;Logical EOF flag
```

The following byte is recognized as the logical EOF character if the above logical EOF flag byte is nonzero.

```
0355   853    DB     1AH        ;EOF character
```

The following byte is the exit code returned by muLISP if the function SYSTEM is called without arguments or if the break option "System" is chosen.

```
0356   854    DB     0          ;Default exit code
```

The following byte stores the computer type number from the following table. It is used by the muLISP console screen and graphics functions.

0 = Unknown computer type	5 = Zenith Z-100 or VT-52
1 = Generic MS-DOS computer	6 = Hewlett-Packard HP-150
2 = IBM PC or "look-alike"	7 = Hewlett-Packard HP-110
3 = ANSI or VT-100 Terminal	8 = NEC APC or ADM-3A
4 = TI Professional Computer	9 = NEC PC-9801

```
0357   855    DB     0          ;Computer type byte
0358   856    DS     4          ;Reserved space, do not use
```

If the following byte is nonzero, the muLISP pseudo-code compiler is included in COM files created by the function SAVE. Only COM files *without* the pseudo-code compiler can be sold as muLISP Runtime Systems. Contact the Soft Warehouse, Inc. for licensing details.

```
035C   860    DB     0          ;Preserve D-code compiler flag
```

The following string is the muLISP read-eval-print loop prompt.

```
035D   861    DB     '$ ',0,0,0,0 ;muLISP prompt string
```

The following string is the default source file name extension recognized by the function LOAD.

```
0363   867    DB     'LSP'      ;Default source file name extension
```

If the following byte is nonzero and muLISP is running on an IBM PC or compatible computer, the cursor temporarily becomes a full block during garbage collections and data space reallocations.

```
0366   870    DB     -1         ;GC cursor indication flag
```

If the following byte is nonzero, input of a character whose high order bit is set is recognized as the first byte of a two-byte KANJI character.

```
0367   871    DB     0          ;Recognize KANJI characters flag
```

If the following byte is nonzero, the header message (see Chapter 2) is displayed when muLISP begins execution.

```
0368   872    DB     -1         ;Display logon header message flag
```

The following character is used by the functions PLOT-DOT, PLOT-LINE, and PLOT-CIRCLE for plotting points when in a text video mode (see Section 5.21).

```
0369   873    DB      '.'      ;Dot used for char-mode graphics
```

The following characters are used by the line-editor to indicate that more text is to the right and/or left of the displayed part of the line.

```
036A   874    DB      '<'      ;More left indicator character
036B   875    DB      '>'      ;More right indicator character
```

If the following byte is nonzero and muLISP is running on an IBM PC or compatible computer in a graphics video mode, muLISP uses its own internal character generator table to display characters 128 through 255. The table located from 400H to 7FFH can be modified by as desired.

```
036C   876    DB      -1       ;Use internal char generator flag
```

The following byte is used as a counter by the function TONE to determine the duration of the tone it produces. If the byte is zero, when muLISP starts up it uses the system clock to determine an appropriate value for the byte.

```
036D   877    DB      0        ;Timing counter byte
```

If one or more of the following words are nonzero, the size of the muLISP console window (see Section 5.21) is limited accordingly.

```
036E   878    DW      0        ;Limit on base row
0370   880    DW      0        ;Limit on base column
0372   882    DW      0        ;Limit on rows
0374   884    DW      0        ;Limit on columns
```

The following byte is the default color used by the functions PLOT-DOT, PLOT-LINE, and PLOT-CIRCLE for plotting points.

```
0376   886    DB      15       ;Default plot color
```

Each entry of the following **Line Editor Control Key Table** points to the machine language routine called by the muLISP line editor when a control key is pressed. No action is taken if the pointer is 0.

```
0380   896    DW     0              ;CTRL-@
0382   898    DW     LFTWRD         ;CTRL-A
0384   900    DW     0              ;CTRL-B
0386   902    DW     0              ;CTRL-C
0388   904    DW     RHTCHR         ;CTRL-D
038A   906    DW     0              ;CTRL-E
038C   908    DW     RHTWRD         ;CTRL-F
038E   910    DW     DELCHR         ;CTRL-G
0390   912    DW     LFTCHR         ;CTRL-H or BACKSPACE
0392   914    DW     RHTTAB         ;CTRL-I or TAB
0394   916    DW     0              ;CTRL-J or LINEFEED
0396   918    DW     0              ;CTRL-K
0398   920    DW     0              ;CTRL-L
039A   922    DW     0              ;CTRL-M or RETURN
039C   924    DW     0              ;CTRL-N
039E   926    DW     0              ;CTRL-O
03A0   928    DW     ESCMOD         ;CTRL-P
03A2   930    DW     CMDCHD         ;CTRL-Q
03A4   932    DW     0              ;CTRL-R
03A6   934    DW     LFTCHR         ;CTRL-S
03A8   936    DW     DELWRD         ;CTRL-T
03AA   938    DW     UNDLIN         ;CTRL-U
03AC   940    DW     INSTOG         ;CTRL-V
03AE   942    DW     0              ;CTRL-W
03B0   944    DW     0              ;CTRL-X
03B2   946    DW     DELLIN         ;CTRL-Y
03B4   948    DW     0              ;CTRL-Z
03B6   950    DW     0              ;CTRL-[ or ESC
03B8   952    DW     0              ;CTRL-\
03BA   954    DW     0              ;CTRL-]
03BC   956    DW     0              ;CTRL-^
03BE   958    DW     DELLFT         ;CTRL-_
03C0   960    DW     DELLFT         ;RUBOUT or DELETE
```

Each entry of the following **Line Editor Extended Function Key Table** contains the second byte returned by an extended function key (see Appendix A) followed by a pointer to the machine language routine called by the muLISP line editor when the extended function key is pressed. The first word of the table contains the number of entries in the table. There is enough room in the table for ten extended function key routines.

```
03C2  962   DW   7          ;Entries in table
03C4  964   DB   83         ;Del
03C5  965   DW   DELCHR     ;Delete char under cursor
03C7  967   DB   75         ;<—
03C8  968   DW   LFTCHR     ;Move left a char
03CA  970   DB   77         ;—>
03CB  971   DW   RHTCHR     ;Move right a char
03CD  973   DB   115        ;Ctrl <—
03CE  974   DW   LFTWRD     ;Move left a word
03D0  976   DB   116        ;Ctrl <—
03D1  977   DW   RHTWRD     ;Move right a word
03D3  979   DB   61         ;F3
03D4  980   DW   UNDLIN     ;Undelete last deleted line
03D6  982   DB   82         ;Ins
03D7  983   DW   INSTOG     ;Insert/replace mode toggle
03D9  985   DB   0
03DA  986   DW   0          ;Free space
03DC  988   DB   0
03DD  989   DW   0
03DF  991   DB   0
03E0  992   DW   0
03E2  994   DB   0
03E3  995   DW   0
03E5  997   DB   0
03E6  998   DW   0
```

C. muLISP GLOBAL STORAGE AREA

Appendix C is an annotated assembly language listing of th muLISP global storage area. It shows the offset address of the variable that delimit the current muLISP data spaces and other system variable The first two columns in each row of the listing are the hexadecimal an decimal offsets in the data segment of the variables.

```
                ORG     0               ;Object space variables
                BASSYM  EQU     100H    ;Base of symbols
0000    0       ENDSYM  DW      ?       ;End of symbols
0002    2       ENDNUM  DW      ?       ;End of numbers
0004    4       ENDATM  DW      ?       ;End of atom space
0006    6       ENDCNS  DW      ?       ;End of conses
0008    8       MAXCNS  DW      ?       ;Max size of cons space

000A    10              DW      ?       ;Reserved
000C    12              DW      ?       ;Reserved
000E    14              DW      ?       ;Reserved

                                        ;Segment pointer variables
0010    16      DEFSEG  DW      ?       ;Function/length cell seg
0012    18      PNPSEG  DW      ?       ;Print name/vector cell seg
0014    20      PNSSEG  DW      ?       ;Print name string segment
0016    22      VECSEG  DW      ?       ;Number vector segment

0018    24              DW      ?       ;Reserved
001A    26              DW      ?       ;Reserved
001C    28              DW      ?       ;Reserved
001E    30              DW      ?       ;Reserved
0020    32              DW      ?       ;Reserved

                                        ;Stack segment variables
                BASSTK  EQU     400H    ;Base of stack space
0022    34      BASCOD  DW      ?       ;End of stk/base of D-code
0024    36      ENDCOD  DW      ?       ;End of D-code
0026    38      MAXCOD  DW      ?       ;Maximum size of D-code
```

```
                                                ;Pname segment variables
0028   40      ENDSTR   DW    ?                ;End of print name strings
002A   42      MAXSTR   DW    ?                ;Max size of string space

                                                ;Vector segment variables
002C   44      ENDVEC   DW    ?                ;End of number vectors
002E   46      MAXVEC   DW    ?                ;Max size of vector space

                                                ;File control block variables
0030   48      BASFCB   DW    ?                ;Base of FCB space in CS
0032   50      IFCB     DW    ?                ;SIF FCB pointer
0034   52      OFCB     DW    ?                ;SOF FCB pointer
0036   54      ENDFCB   DW    ?                ;End of FCB space in CS

0038   56      THRVAL   DW    ?                ;Thrown value (0 = inactive)
003A   58      PRECSN   DW    ?                ;Current precision
003C   60      UNDFLO   DW    ?                ;Current underflow

003E   62      GCCTR    DW    ?                ;Garbage collection counter
0040   64      RACTR    DW    ?                ;Reallocation counter

0042   66      ORICS    DW    ?                ;Original code segment
```

44
46
48
4A ATOMFREEMINI } devant provoquer un GC [Valeurs de ATOMFREE
4C VECTORFREEMINI } (9) Vectorfree après GC
4E CONSFREE minimi Consfree (mini en 8 1D9)
50 temporaire : Sauvegarde de AX
52 Position dans table de gestion des erreurs numériques/routines
54
56 Nil si on saisit un entier, T si réel (.9) mini 5CB3 | Aigue par défaut
58 fraction 1
5A fraction 2
5C
5E longueur courante du Prompt de travail
60
62

D. SAMPLE ASSEMBLY LANGUAGE ROUTINE

Appendix D is an annotated assembly language listing of the function COPY that was discussed in Chapter 9.

```
        ASSUME CS:CSEG
CSEG    SEGMENT

ENDATM  EQU     04H             ;Offset of ENDATM in DS
CONS    EQU     0B27H           ;Offset of CONS in CS

                                ;FUNCTION: (COPY OBJ)
COPY:   JCXZ    RETNIL          ;Return NIL if no arguments
        MOV     SI,[BP]         ;SI: OBJ
COPYSI: CMP     SI,WORD PTR DS:[ENDATM]
        JB      COPY1           ;Jump if (ATOM OBJ)
        MOV     [BP],SI         ;Push OBJ on BP stack
        INC     BP
        INC     BP
        MOV     SI,[SI]         ;SI: (CAR OBJ)
        EVEN
        CALL    COPYSI          ;DI: (COPY (CAR OBJ))
        XCHG    DI,[BP-2]       ;Exchange OBJ with stack top
        MOV     SI,ES:[DI]      ;SI: (CDR OBJ)
        EVEN
        CALL    COPYSI          ;DI: (COPY (CDR OBJ))
        DEC     BP              ;Restore BP
        DEC     BP
        MOV     [BP+2],DI       ;Store DI on top of stack
        MOV     CX,4            ;Set CX to indicate 2 arguments
        MOV     AX,OFFSET CONS
        JMP     AX              ;DI: Cons copies of CAR & CDR

COPY1:  MOV     DI,SI           ;DI: OBJ
        RET

RETNIL: MOV     DI,100H         ;DI: NIL
        RET
CSEG    ENDS
        END     COPY
```

E. WORDSTAR-LIKE EDITOR COMMANDS

Appendix E summarizes the control key commands available when the muLISP editor is operated in WordStar-like mode. The commands are divided into the same categories used in Chapter 7.

Normal Key	Alternate	IBM Key	Cursor control commands
CTRL-D		→	move forward a char
CTRL-S		←	move backward a char
CTRL-F		CTRL →	move forward a word
CTRL-A		CTRL ←	move backward a word
CTRL-I	TAB	→|	move right to tab stop
CTRL-Q I			move left to tab stop
CTRL-Q D			move to right end of line
CTRL-Q S			move to left end of line
CTRL-E		↑	move up a line
CTRL-X		↓	move down a line
CTRL-Q E		Home	move to top row
CTRL-Q X		End	move to bottom row
CTRL-M	RETURN	←┘	move to start of next line
CTRL-J	LINEFEED	CTRL ←┘	move down line and indent
CTRL-Q P			move to last position

Normal Key	Alternate	IBM Key	Display control commands
CTRL-W			scroll text down a line
CTRL-Z			scroll text up a line
CTRL-R		PgUp	scroll text down a screen
CTRL-C		PgDn	scroll text up a screen
CTRL-Q C		CTRL-End	move to end of text
CTRL-Q R		CTRL-Home	move to start of text

Normal Key	Alternate	IBM Key	Entering text commands
CTRL-V		Ins	insert/replace mode toggle
CTRL-N			insert a new line
CTRL-P			escape char

Normal Key	Alternate	IBM Key	Deleting text commands
CTRL-G		Del	delete char under cursor
CTRL-H	BACKSPACE	<——	delete char left of cursor
CTRL-_	RUBOUT	CTRL <- _	delete char left of cursor
CTRL-T			delete word under cursor
CTRL-Q T			delete word left of cursor
CTRL-Y			delete line under cursor
CTRL-Q Y			delete right end of line
CTRL-Q RUBOUT			delete left end of line
CTRL-U			restore last deleted line

Normal Key	Alternate	IBM Key	Block commands
CTRL-O B	CTRL-K B		mark beginning of block
CTRL-O E	CTRL-K K		mark end of block
CTRL-O M	CTRL-K V		move block to cursor
CTRL-O C	CTRL-K C		copy block at cursor
CTRL-O D	CTRL-K Y		delete block
CTRL-O W	CTRL-K W		write block to file
CTRL-O J B	CTRL-Q B		move to beginning of block
CTRL-O J E	CTRL-Q K		move to end of block

Normal Key	Alternate	IBM Key	Search & replace commands
CTRL-Q F			search for string
CTRL-Q A			search and replace string
CTRL-L			find/replace again
CTRL-Q L			find & replace all remaining
CTRL-Q V			move to last find/replace

Normal Key	Alternate	IBM Key	List structure commands
CTRL-^ D		ALT-D	move forward an S-expr
CTRL-^ S		ALT-S	move backward an S-expr
CTRL-^ C		ALT-C	move forward to next defn
CTRL-^ R		ALT-R	move to beginning of defn
CTRL-^ T		ALT-T	delete S-expr
CTRL-^ !		ALT-!	evaluate S-expr

Normal Key	Alternate	IBM Key	File I/O commands
CTRL-K D			save file & clear edit buffer
CTRL-K S			save file
CTRL-K A	CTRL-K Q		quit file & clear edit buffer
CTRL-K R			merge file

F. EMACS-LIKE EDITOR COMMANDS

Appendix F summarizes the control key commands available when the muLISP editor is operated in EMACS-like mode. The commands are divided into the same categories used in Chapter 7.

Normal Key	Alternate	IBM Key	Cursor control commands
CTRL-F		→	move forward a char
CTRL-B		←	move backward a char
CTRL-X F	ALT-F	CTRL →	move forward a word
CTRL-X B	ALT-B	CTRL ←	move backward a word
CTRL-I	TAB	→|	move right to tab stop
CTRL-E			move to right end of line
CTRL-A			move to left end of line
CTRL-P		↑	move up a line
CTRL-N		↓	move down a line
CTRL-M	RETURN	↵	move to start of next line
CTRL-J	LINEFEED	CTRL ↵	move down line and indent

Normal Key	Alternate	IBM Key	Display control commands
CTRL-X V	ALT-V	PgUp	scroll text down a screen
CTRL-V		PgDn	scroll text up a screen
CTRL-X >		End	move to end of text
CTRL-X <		Home	move to start of text

Normal Key	Alternate	IBM Key	Entering text commands
CTRL-X I	ALT-I		insert/replace mode toggle
CTRL-O		Ins	insert a new line
CTRL-Q			escape char

Normal Key	Alternate	IBM Key	Deleting text commands
CTRL-D		Del	delete char under cursor
CTRL-X D	ALT-D		delete word under cursor
CTRL-H	BACKSPACE	←	delete char left of cursor
CTRL-_	RUBOUT	CTRL ←	delete word left of cursor
CTRL-K			delete right end of line
CTRL-X K	ALT-K		delete left end of line
CTRL-Y			restore last deleted line

Normal Key	Alternate	IBM Key	Block commands
CTRL-U B	CTRL-@	CTRL-?	mark beginning of block
CTRL-U E			mark end of block
CTRL-U M			move block to cursor
CTRL-U C			copy block at cursor
CTRL-U D	CTRL-W		delete block
CTRL-X W	ALT-W		write block to file

Normal Key	Alternate	IBM Key	Search & replace commands
CTRL-S			search for string
CTRL-X %	ALT-%		search and replace string
CTRL-L			find/replace again
CTRL-X L			find/replace all remaining

Normal Key	Alternate	IBM Key	List structure commands
CTRL-Z F	CTRL-Z)		move forward an S-expr
CTRL-Z B	CTRL-Z (		move backward an S-expr
CTRL-Z]			move forward to next defn
CTRL-Z [			move to beginning of defn
CTRL-Z K			delete S-expr
CTRL-Z E	CTRL-Z !	ALT-!	evaluate S-expr

Normal Key	Alternate	IBM Key	File I/O commands
CTRL-X S	ALT-S		save file
CTRL-X A	ALT-A		quit file & clear edit buffer
CTRL-X R	ALT-R		merge file

G. muLISP-86 TO '87 SOURCE FILE CONVERSION

The muLISP file and I/O systems have been extensively reworked for the '87 version of muLISP. In addition to a number of new file system functions, in the spirit of the Common LISP naming convention many of the existing functions and control variables have been renamed to be more mnemonic. Therefore, source files for muLISP-86 programs that use these functions will need to be modified to run correctly under muLISP-87.

Modifying muLISP-86 source files for muLISP-87 is relatively easy using the "find-and-replace" command provided by the muLISP editor (see Chapter 7). The first step is to make the replacements

(RDS) —> (CLOSE-INPUT-FILE)
(RDS *filename*) —> (OPEN-INPUT-FILE *filename*)
(WRS) —> (CLOSE-OUTPUT-FILE)
(WRS *filename*) —> (OPEN-OUTPUT-FILE *filename*)
(WRS *filename flag*) —> (OPEN-OUTPUT-FILE *filename flag*)
RDS —> *INPUT-FILE*
WRS —> *OUTPUT-FILE*
(INPUTFILE) —> (INPUT-FILE)
(INPUTFILE *filename*) —> (INPUT-FILE-P *filename*)
(OUTPUTFILE) —> (OUTPUT-FILE)
(OUTPUTFILE *filename*) —> (OUTPUT-FILE-P *filename*)
(OPENFILES) —> (APPEND (INPUT-FILES) (OUTPUT-FILES))
(READPTR *n*) —> (FILE-READ-POSITION *n*)
(READPTR 'EOF) —> (FILE-LENGTH (INPUT-FILE))
(WRITEPTR *n*) —> (FILE-WRITE-POSITION *n*)
(WRITEPTR 'EOF) —> (FILE-LENGTH (OUTPUT-FILE))
(RATOM) —> (READ-ATOM)
ECHO —> *INPUT-ECHO* or *OUTPUT-ECHO*
(PRINT *object*) —> (WRITE *object*)
(LINELENGTH *n*) —> (FILE-LINE-LENGTH *n*)
LINELENGTH —> *AUTO-NEWLINE*

In muLISP-86 the functions READ-CHAR and RATOM assigned the value they returned to the variable RATOM. If a muLISP 8/ program uses this feature, make the replacements

(READ-CHAR) ⟶ (SETQ RATOM (READ-CHAR))
(READ-ATOM) ⟶ (SETQ RATOM (READ-ATOM))

In muLISP-86 the control variable READ-CHAR determined whether the input functions used line edit mode or raw input mode for console input. In muLISP-87 all input functions use line edit mode for console input, except the function READ-BYTE which uses raw input mode. If a call to the function READ-CHAR in a muLISP-86 program used raw input mode, make the replacement

(READ-CHAR) ⟶ (ASCII (READ-BYTE))

The source file MULISP86.LSP provides an alternative to making the above transformations in muLISP-86 source files. By defining muLISP-86 functions such as RDS and WRS that are no longer supported it makes muLISP-87 "look" like muLISP-86. After loading MULISP86.LSP, most muLISP-86 source files will run correctly under muLISP-87 with only minimal modification, if any. Although this alternative is the quickest way to get up and running, it does not fully utilize the new features of muLISP-87.

The algorithm muLISP-87 uses for rounding ratios is faster, yields a flatter percentage error curve, and provides the user with twice the selectivity over the precision level (see Section 5.12). Thus, precision settings given to the function PRECISION should be twice minus one what was used in muLISP-86. Also, instead of selecting exact mode arithmetic by the call (PRECISION 0), set the control variable *EXACT-MODE* to T.

CONCEPT INDEX

Abort option 4-9 5-261
absolute value 5-141
active display page 5-284
actual arguments 5-212
address typing 9-2
alphabetic characters 5-91
Animal game 1-8,10 2-9 3-1
arc cosine 5-139
arc sine 5-139
arc tangent 5-139
argument passing 9-5
arithmetic 5-110
arithmetic functions 5-124:133
array package 6-1:5
  array dimension 6-4
  array rank 6-4
  array options 6-2
  arrays 6-1
  Dimension Error 6-2
  vector 6-2
Artificial Intelligence
  Books 10-3
  Language *iii*
  Periodicals 10-2
  Professional societies 10-1
ASCII character 5-95
assembly language routine A-13
assignment functions 5-69:75
association list 5-16,17,22,36
atom space 4-6 9-2
atoms 5-3
attribute 5-80
auto-quoting 2-6 4-2
auxiliary functions 5-2

backup, diskette 1-12
base column 5-277
base page 9-1 5-269 A-2:10
base row 5-277
big integers 4-3 5-62
binary trees *iii* 4-4 5-3
bit-wise logical functions 5-150:153
blank characters 5-184
bold face type 5-1
Boolean operations 5-66
bracket 5-172
break characters 5-161
Break option 4-9 5-260
break package 4-9 5-260
break-points 5-261

car branch 5-3
car cell 4-1,4 5-3
case conversion 5-97,106,175,185
cdr branch 5-4
cdr cell 4-1,4 5-4
chain load 2-9
chronological ordering 5-65
circular list 5-41,228
clear buffer 5-166
closure package 6-29:34
  dynamic closure 6-29
  generator 6-29
color graphics adapter 5-279,281,283
color graphics monitor 5-281,283,285
COM files 1-10 5-209
command processor 5-249
comments 5-169,173,244
Common Lisp *iv* A-15
comparator functions 5-61:65
compilation 5-82 8-1
compile time 5-225
compiler, muLISP 8-1:20
  basic procedure 8-7:10
  errors 8-15,16
  file compiler 8-1

functions 8-3:6
improved performance 8-11:14
incremental compiler 8-1
perfect vs. compatible 8-17
potential problems 8-17:20
COMPRESS 5-99
computer reset 4-11
computer type 5-276 A-7
COMTOEXE utility 1-10 5-209
COND, implicit 2-6 5-232,234,235
condensing 5-89
conditional fork 5-232
conses 4-4 5-3,18 9-2
console interrupt 4-11 5-222
console screen functions 5-276:287
console window 5-277
constants 5-84,215
constructor functions 5-18:37
Continue option 4-9 5-260
control characters 2-4 5-95
control constructs 5-232:249
Control Key Table A-9
control stack 4-6
control variables 5-1
copyright notice *ii* 1-12
cosine 5-138
current directory 5-189
cursor 4-8 5-279,285 7-6

D-code 4-6 5-82,84,87
data compaction 4-6
data objects 4-1
data representation 9-2
data space 4-1
data space partition 4-6
data space reallocation 4-8
data structures 4-1:15
data types 4-6
debug package 7-1
backtrace 7-7
break-point facility 7-7
loading 7-1
statistics facility 7-12
trace facility 7-2
trace history 7-5
decimal notation 5-110,188
de-compilation 5-82
default drive 5-189
Macro Character Table A-4
Default Read Table A-2
defflavor options 6-12,13
definition functions 5-82:89
defstruct options 6-7:9
demonstration programs 3-1
destructuring 5-88
device names 5-189
dimension, array 6-4
Dimension Error 4-12 6-2
directory entry 5-198,209
Disk Full error 4-12 5-177,256
diskette backup 1-12
distilled code 4-6 5-82,84,87
DOCTOR program 3-1
dollar sign prompt 5-2
dot notation 4-4 5-178
dotted-pairs 4-4
double precision 5-111
driver function 5-256
dynamic closure 6-29
dynamic memory management *iii*

echo 5-175,184,185
editor, muLISP 7-1
editor, nonresident 7-1
Eights Puzzle 3-2
EMACS-like editor commands A-16
empty list 4-5
End-Of-File 4-12 5-159,163,204
end of file, test for 5-165
enhanced graphics adapter 5-281,283
enhanced display monitor 5-283,285
environment 2-10 5-209

environment functions 5-209:211
equality 5-63
error break 4-12 5-110,261
error messages 4-12:16
escape character 5-160,186
eval function 5-83,212,217 9-5
eval-LISP driver 2-3
eval-quote LISP 5-259
evaluation functions 5-212:231
Exact Mode 4-12 5-134
EXE files 5-209
exit code 5-248,249 A-6
exiting muLISP 2-2 5-248
expansion, macro 5-223:225
EXPLODE function 5-98
extended COND 2-6
Extended Function Key Table A-10
extended function keys A-1

Factorial function 2-8
false, truth value 5-66
file backup feature 5-198
file compiler 8-1
file line length 5-184,205
File not found 2-11 4-13
file system primitives 5-189:211
flag functions 5-80,81
flags 4-2 5-80
flavors package 6-10:17
  defflavor options 6-12,13
  flavor 6-10
  initable instance vars 6-12
  instance variables 6-10
  Keyword Not Handled 6-16
  method combination 6-10
  methods 6-10
  Missing Required Flavor 6-12,16
  object 6-10
  Unclaimed Message 6-17
  whoppers 6-11
floating point numbers 5-56
floating slash arithmetic 5-110
formal arguments 5-212
forms 5-213,234
full condensing 5-89
function body 2-6 5-213,232,235
function cell 4-2
function definition 5-83,212
function type 5-212,213
functions 5-1,215

garbage collection 4-7 5-37,250:253
GC statistics 5-251
generator 6-29
generic MS-DOS 5-276
global storage area 9-1 5-269 A-11
glossary 11-1:7
graphics display monitor 5-283,285
graphics package 6-18:22
greatest common divisor 5-131

Hanoi, Tower of 3-2
hard copy printout 5-185,198 7-25
hardware functions 5-266:275
header message 2-1 5-276
help facility 3-3
hexadecimal numbers 5-251
home position 5-277

IBM Personal Computer
  color graphics display 5-281:285
  computer type 5-276
  extended function key A-1
  hardware requirements 1-11
  keyboard notation 1-14
  monochrome display 5-281,283,285
identity 5-61,237
identity cell 4-3
if-then-else construct 5-238
implicit COND 2-6 5-232:239
implicit PROGN 2-6 5-232:239
IMPLODE function 5-99

Include Error 4-13 6-9
Incompatible Definition 4-13 6-7
incremental compilation 5-82 8-1,2
indicator key, property 4-2 5-76:79
initable instance variables 6-12
input character macros 5-159,160
input control variables 5-175,176
input functions 5-159:168
input primitives 5-159:176
instance variables 6-10
Insufficient Arguments 4-13 5-110
Insufficient Code Space 8-16
Insufficient Memory 2-11 4-13
integers 4-3
interactive environment *iii*
interrupt character 4-11
interrupt, console 4-11
interrupt message 4-11
interrupt vectors 5-273
interval 5-119,121
invalid argument errors 4-12
Invalid Argument 4-14 5-135:137
Invalid Place Form 4-14 5-73,130
inverse video 5-281
irrational functions 5-134:140
isomorphic structures 5-63
iteration 2-8 5-235

Kanji characters A-7
Keyword Not Handled 4-14 6-16

LAMBDA type function 5-87,212
least common multiple 5-133
length cell 4-3
lexicographical comparison 5-93,94,102:104
library source files 5-1
line edit mode 5-159
line editing 2-4 5-159
line length 5-184,205
LISP *iii* 2-7 10-1
LISP bibliography 10-3
list merge sort 5-54
list notation 4-4 5-178
LOAD command 2-9,10 5-211
local variables 2-7 5-69
logarithm 5-136
logical functions 5-66:68
LOGO 6-18
lowercase characters 5-90
lowercase conversion 5-97,106,185

machine language routine
  address of 5-83
  argument passing 9-5
  avoiding 5-266 9-1
  base page 9-1 A-2:10
  calling convention 9-7
  data representation 9-2
  external 5-273
  eval function 9-5
  function 9-5
  global storage area 9-1 A-11
  linking 5-84 9-9
  loading 9-9
  reasons for 9-1
  sample A-13
  segments 9-3
  special form 9-5 A-2
  writing 9-1
macro characters 5-160,169:174
macro expansion 5-212
macro expansion functions 5-223:225
macros 5-1,88,212
mapping functions 5-226:228
mapping predicates 5-229:231
Memory Full 4-14 5-251,253
memory management
  data space reallocation *iii* 4-8
  functions 5-250:255
  garbage collection 4-7 5-37,250:253

initial data space partition 4-6
statistics 5-251
merge sort 5-54
METAMIND game 3-3
method combination 6-10
methods 6-10
Microsoft mouse 6-23
miscellaneous functions 5-288:290
Missing Required Flavor 4-14 6-16
modifier functions 5-38:54
modulus 5-148
monochrome display adapter 5-275,281,283
monochrome monitor 5-283,285
mouse package 6-23:28
mouse cursor display 6-24
mouse parameters 6-23
MS-DOS 1-10 5-189,248,273 A-6
muLISP
array package 6-1:5
base page 9-1 A-2:10
closure package 6-29:34
compiler 8-1:20
conversion from '86 A-18
copyright notice 1-12
demonstration programs 1-8 3-1
design philosophy *iv*
editor 7-2
environment 2-10 5-209
error messages 4-12:16
flavors package 6-10:17
global storage area A-11
header message 2-1
interpreter 4-1:16
lessons 3-4
line editor 2-4 5-159
machine language kernel 1-3:6
machine language routines 9-1:10
major features 1-3:9
master diskette 1-10
memory management
mouse package 6-23:28
operating procedures 2-1:11
package library 1-7 6-1:34
program development sys 1-6 7-1
programming 2-6:8
prompt 2-1 5-257
runtime system 1-6 5-210
serial number 2-1
source files 2-9
structure package 6-6:9
system contents 1-10
system requirements 1-11
turtle graphics package 6-18:22
Tutorial System 1-9 3-4 5-196
version date 2-1
version number 2-1
muLISP Editor
block commands 7-11
command functions 7-19
command summary A-14,16
cursor control commands 7-6
Edit option 7-3
EMACS-like 7-2, A-16
entering text 7-9
file saving commands 7-15
global constants 7-22
global variables 7-21
Lisp option 7-4
list structure commands 7-13
loading editor 7-2
point 7-17
Print option 7-3
Quit option 7-4
saving editor 7-2
Screen option 7-4
search and replace 7-12
text deletion commands 7-10
theory of operation 7-17
top-level menu 7-3
window commands 7-16

named structures 6-6
newline character 5-179,181,184
NIL 4-5 5-55,58,61,66,215
NLAMBDA function 5-83,87,212
no-eval function 5-212,214,217
no-spread function 5-213
Non-eval Function 8-15
Noninteger Argument 4-15 5-110
non-local exit 1-5
Nonnumeric Argument 4-15 5-110
nonrecursive loop 2-7 5-235
Nonsymbolic Argument 4-15 5-70,129 6-7 8-15
normal video 5-281
null list 4-5
number vector 4-4
numbers 4-3 9-2
numeric characters 5-91
numeric weights 5-92
numerical comparators 5-117:123
numerical error trap handlers 5-157,158
numerical functions 5-110:158
numerical recognizers 5-114:116
numerical selectors 5-141:149

object 6-10
object list 5-37
option prompt 4-9 5-260
output control vars. 5-184:188
output functions 5-177:183
output primitives 5-177:188

pi 5-140
place form 5-73,130
pname cell 4-2
point, editor's 7-17
pointer cells 4-1
pointer space 4-6 5-253 9-2
precision 5-111,112
predicate 5-55,61,234,239
primitives 5-1
print name functions 5-90:109
print name string 4-2
print names 4-2 5-90
printout, hard-copy 5-185,198
Problem Report, muLISP *ii*
PROGN, implicit 2-6 5-235,236,239
programming fundamentals 2-6:8
prompt, muLISP 2-1 5-257
property functions 5-76:79
property list 4-2 5-76
property list cell 4-2
property value 5-76
pseudo-compilation 5-82
pseudo-registers A-4

radix base 5-175,187
random numbers 5-154:156
rank, array 6-4
rational arithmetic 5-110
rational numbers 5-56
ratios 4-3 5-62
raw input mode 5-159
read-eval-draw loop 6-22
read-eval-print loop 2-2 5-257
read position 5-196
reading source files 2-9
readtable 5-170 A-3
real time systems 4-8
recognizer functions 5-55:60
record structures 6-6
recursion *iii* 2-6
remainder 5-148
reset, computer 4-11
resident editor 7-1
Restart option 4-10 5-261
return code 5-248
run time 5-225
runtime system 1-6 5-210

SAVE command 2-10 5-209
searcher functions 5-3:17
segments 9-3
selector functions 5-3:17
series approximation 5-113
side-effects 5-38,69
sign cell 4-3
sine 5-138
single precision 5-111
slash notation 5-110,188
slot description 6-7
small integers 4-3 5-61
Soft Warehouse, Inc. *ii* 1-7 5-210
sorting 5-54
source files 2-9
space characters 5-184
Special Form Jump Table A-2
special forms 5-1,214 9-5
spread function 5-212
stable merge 5-53
stable sort 5-54
stack space 4-6
stack operations 5-74,75
standard input file (SIF) 5-190,195
standard input source (SIS) 5-159,190
standard output file (SOF) 5-197:203
standard output sink (SOS) 5-177,197
string functions 5-98:108
structure package 6-6:9
  defstruct options 6-7,8
  Include Error 6-9
  Incompatible Definition 6-7
  named structures 6-6
  Nonsymbolic Argument 6-7
  slot description 6-7
  Unnamed Structure 6-8
  unnamed structures 6-6
  Unsupported Option 6-7
super-parenthesis 5-172
symbol creation 5-109
symbol list 5-37
symbols 4-1 9-2
Syntax Error 4-15
SYS files 2-10 5-209:211
system calls, MS-DOS 5-273
system contents 1-10
system load & run 2-1
System option 4-10 5-261
system requirements 1-11

T 5-55,61,66,215
tab character 5-184
tangent 5-138
thrashing 4-8 5-251
time 5-289
token 5-160
top-level loop 5-256,259,261
Top-level option 4-9 5-261
Tower of Hanoi Puzzle 3-2
trace facility 7-2
trace history 7-5
trademarks *ii*
transcendental functions 5-134:140
true, truth value 5-66
truth values 5-66
turtle graphics package 6-18:22
  colors 6-20
read-eval-draw loop 6-22
type checking 4-1 9-2

Unclaimed Message 4-15 6-17
Undefined Function 4-15 5-217,222,253
Undefined Function Call 8-15
Undefined Structure 4-15 6-8
underflow 5-113
underline 5-281
Unnamed Structure 4-15 6-8
unnamed structures 6-6
Unsupported Option 4-16 6-7
uppercase characters 5-90

ppercase conversion 5-97,106,175
er interface functions 5-256:265

alue cell 4-2
riable stack 4-6 5-37
ctor 6-2
ctor cell 4-4
ctor space 4-6
rsion number, date 2-1
sual display page 5-284

hitespace characters 5-160,161
hoppers 6-11
ild-card filname characters 5-208
indows 5-277,278
ordStar-like commands A-16
rite position 5-204

ero Divide error 4-16 5-110

)86 code segment 5-83 9-3
)86 microprocessor 1-11
5-268,273 9-1

FUNCTION INDEX

This is an index of the functions, special forms, macros, control variables, and constants in muLISP. The brackets following the name of functions and special forms enclose a list of names denoting the function's or form's arguments. All entries are followed by the primitive's category, type, and the page on which the primitive is described.

Name [arguments]	Category	Type	Page
ABS [*n*]	Numerical	Function	5-141
ABSTRAP [*object*]	Numerical	Function	5-158
ACONS [*key*, *object*, *alist*]	Constructor	Function	5-36
ACOS [*n*]	Numerical	Function	5-139
ADD1 [*n*]	Numerical	Function	5-128
ADJOIN [*object*, *list*, *test*]	Constructor	Function	5-33
ALLOCATE [*n*]	Memory	Function	5-254
ALPHA-CHAR-P [*symbol*]	Character	Function	5-91
ALPHANUMERICP [*symbol*]	Character	Function	5-91
AND [$form_1$, ... $form_n$]	Logical	Spcl form	5-67
APPEND [$list_1$, ... $list_n$]	Constructor	Function	5-21
APPLY [*function*, arg_1, ..., *list*]	Evaluator	Function	5-219
AREF [*array*, $subscript_1$, ...]	Array	Function	6-3
ARRAY-DIMENSION [*array*, *axis*]	Array	Function	6-4
ARRAY-DIMENSIONS [*array*]	Array	Function	6-4
ARRAY-IN-BOUNDS [*array*, sub_1, ...]	Array	Function	6-5
ARRAY-RANK [*array*]	Array	Function	6-4
ARRAY-TOTAL-SIZE [*array*]	Array	Function	6-4
ARRAYP [*object*]	Array	Function	6-5
ASCII [*atom*]	Character	Function	5-95
ASH [*n*, *count*]	Numerical	Function	5-152
ASIN [*n*]	Numerical	Function	5-139
ASSOC [*key*, *alist*, *test*]	Searcher	Function	5-16
ASSOC-IF [*test*, *alist*]	Searcher	Function	5-16
ATAN [*n*, *m*]	Numerical	Function	5-139
ATOM [*object*]	Recognizer	Function	5-57
AUTO-NEWLINE: T	Output	Ctrl var.	5-184

BACK [*distance*]	Graphics	Function	6-21
BACKGROUND-COLOR [*n*]	Screen	Function	5-281
BINARY-LOAD [*filename*, *offset*]	Hardware	Function	5-274
BLANK-COMPRESS: NIL	Output	Ctrl var.	5-184
BLINK: NIL	Screen	Ctrl var.	5-287
BORDER-COLOR [*n*]	Screen	Function	5-282
BREAK [*object*, *message*]	User interface	Function	5-260
BREAK	User interface	Variable	5-260
BUTLAST [*list*, *n*]	Constructor	Function	5-26
BUTTON-PRESSES [*button*]	Mouse	Function	6-27
BUTTON-RELEASES [*button*]	Mouse	Function	6-27
CAAAAR [*object*]	Selector	Function	5-5
CAAADR [*object*]	Selector	Function	5-5
CAAAR [*object*]	Selector	Function	5-5
CAADAR [*object*]	Selector	Function	5-5
CAADDR [*object*]	Selector	Function	5-5
CAADR [*object*]	Selector	Function	5-5
CAAR [*object*]	Selector	Function	5-5
CADAAR [*object*]	Selector	Function	5-5
CADADR [*object*]	Selector	Function	5-5
CADAR [*object*]	Selector	Function	5-5
CADDAR [*object*]	Selector	Function	5-5
CADDDR [*object*]	Selector	Function	5-5
CADDR [*object*]	Selector	Function	5-5
CADR [*object*]	Selector	Function	5-5
CAR [*object*]	Selector	Function	5-5
CASE [*key*, $case_1$, ...]	Control	Macro	5-240
CATCH [*tag*, $form_1$, ...]	Control	Spcl form	5-245
CDAAAR [*object*]	Selector	Function	5-5
CDAADR [*object*]	Selector	Function	5-5
CDAAR [*object*]	Selector	Function	5-5
CDADAR [*object*]	Selector	Function	5-5
CDADDR [*object*]	Selector	Function	5-5
CDADR [*object*]	Selector	Function	5-5
CDAR [*object*]	Selector	Function	5-5

CDDAAR [*object*]	Selector	Function	5-5
CDDADR [*object*]	Selector	Function	5-5
CDDAR [*object*]	Selector	Function	5-5
CDDDAR [*object*]	Selector	Function	5-5
CDDDDR [*object*]	Selector	Function	5-5
CDDDR [*object*]	Selector	Function	5-5
CDDR [*object*]	Selector	Function	5-5
CDR [*object*]	Selector	Function	5-4
CEILING [*n*, *m*]	Numerical	Function	5-115
CHAR [*atom*, *n*]	String	Function	5-101
CHAR-CODE [*symbol*]	Character	Function	5-96
CHAR-DOWNCASE [*symbol*]	Character	Function	5-97
CHAR-EQUAL [sym_1, ... sym_n]	Character	Function	5-94
CHAR-GREATERP [sym_1, ... sym_n]	Character	Function	5-94
CHAR-LESSP [sym_1, ... sym_n]	Character	Function	5-94
CHAR-NOT-EQUAL [sym_1, ... sym_n]	Character	Function	5-94
CHAR-NOT-GREATERP [sym_1, ...]	Character	Function	5-94
CHAR-NOT-LESSP [sym_1, ... sym_n]	Character	Function	5-94
CHAR-UPCASE [*symbol*]	Character	Function	5-97
CHAR= [$symbol_1$, ... $symbol_n$]	Character	Function	5-93
CHAR< [$symbol_1$, ... $symbol_n$]	Character	Function	5-93
CHAR> [$symbol_1$, ... $symbol_n$]	Character	Function	5-93
CHAR<= [$symbol_1$, ... $symbol_n$]	Character	Function	5-93
CHAR>= [$symbol_1$, ... $symbol_n$]	Character	Function	5-93
CHAR/= [$symbol_1$, ... $symbol_n$]	Character	Function	5-93
CLEAR-INPUT [*source*]	Input	Function	5-166
CLEAR-SCREEN []	Screen	Function	5-280
CLOSE-INPUT-FILE [*file*]	File	Function	5-192
CLOSE-OUTPUT-FILE [*file*]	File	Function	5-200
CLOSURE [*varlist*, *function*]	Closures	Function	6-29
CLOSURE-ALIST [*closure*]	Closures	Function	6-34
CLOSURE-FUNCTION [*closure*]	Closures	Function	6-34
CLOSURE-VARIABLES [*closure*]	Closures	Function	6-34
CLOSUREP [*closure*]	Closures	Function	6-32
CLR []	Graphics	Function	6-20
CODE-CHAR [*n*]	Character	Function	5-96

COLUMN []	Screen	Function	5-279
COMMENT [*comments*]	Control	Spcl form	5-244
COMPILE [*name*]	Compiler	Function	8-3
COMPILE-FILE [*cfiles*, *lfiles*]	Compiler	Function	8-6
:CONC-NAME [*prefix*]	Structure	Option	6-7
COND [$cond_1$, ... $cond_n$]	Control	Spcl form	5-239
CONDENSE: NIL	Definition	Ctrl var.	5-89
CONS [obj_1, obj_2]	Constructor	Function	5-18
CONSOLE-BREAK []	Evaluator	Function	5-222
CONSP [*object*]	Recognizer	Function	5-59
CONSTANTP [*object*]	Evaluator	Function	5-215
:CONSTRUCTOR [*name*, *arglist*]	Structure	Option	6-7
:COPIER [*name*]	Structure	Option	6-8
COPY-ALIST [*alist*]	Constructor	Function	5-22
COPY-CHAR-TYPE [chr_1, chr_2, *flg*]	Input	Function	5-170
COPY-CLOSURE [*closure*]	Closures	Function	6-32
COPY-LIST [*list*]	Constructor	Function	5-22
COPY-TREE [*object*]	Constructor	Function	5-22
COS [*n*]	Numerical	Function	5-138
COS-DEG [*angle*]	Graphics	Function	6-18
COUNT [*object*, *list*, *test*]	Searcher	Function	5-11
COUNT-IF [*test*, *list*]	Searcher	Function	5-11
CSMEMORY [*offset*, *value*, *flag*]	Hardware	Function	5-268
CURSOR-LINES [*start-line*, *end-line*]	Screen	Function	5-285
DECF [*place*, *n*]	Numerical	Macro	5-130
DECQ [*symbol*, *n*]	Numerical	Spcl form	5-129
DEFAULT-DRIVE [*drive*]	File	Function	5-208
DEFAULT-PATH [*drive*, *path*]	File	Function	5-208
DEFFLAVOR [*name*, *vars*, *flvs*, opt_1, ...]	Flavors	Macro	6-12
DEFMACRO [*sym*, *arglist*, $form_1$, ...]	Definition	Spcl form	5-88
DEFMETHOD [(*flv type meth*), *args*, $form_1$, ...]	Flavors	Macro	6-14
DEFSTRUCT [(*name opt*$_1$ ...), $slot_1$, ...]	Structure	Macro	6-7
DEFUN [*sym*, *arglist*, $form_1$, ...]	Definition	Spcl form	5-87
DEFUN [*sym*, (*typ arg* $form_1$ ...)]	Definition	Spcl form	5-87
DEFUN [*symbol*, *address*]	Definition	Spcl form	5-87

DEFWHOPPER [(*flv meth*), *args*, $form_1$, ...]	Flavors	Macro	6-15
DELETE [*item*, *list*, *test*]	Modifier	Function	5-44
DELETE-COMPILER []	Compiler	Macro	8-5
DELETE-DUPLICATES [*list*, *test*]	Modifier	Function	5-46
DELETE-FILE [*file*]	File	Function	5-206
DELETE-IF [*test*, *list*]	Modifier	Function	5-44
DELETE-LINES [*n*]	Screen	Function	5-280
DELETE-NTH [*list*, *n*]	Modifier	Function	5-45
DENOMINATOR [*n*]	Numerical	Function	5-143
DENTRAP [*object*]	Numerical	Function	5-158
DIGIT-CHAR-P [*symbol*, *radix*]	Character	Function	5-92
DIRECTORY [*path*]	File	Function	5-208
DISPLAY-PAGE [*n*, *active-only*]	Screen	Function	5-284
DIVIDE [*n*, *m*]	Numerical	Function	5-149
DO [*letlist*, $form_1$, ... $form_n$]	Control	Macro	5-242
DO* [*letlist*, $form_1$, ... $form_n$]	Control	Macro	5-242
DOLIST [(*var list result*), $form_1$, ...]	Control	Macro	5-243
DOS [*command*]	Control	Function	5-249
DOT [*x*, *y*]	Graphics	Function	6-19
DOTIMES [(*var count result*), $form_1$, ...]	Control	Macro	5-243
DRIVER [*file*]	User interface	Function	5-257
DRIVER: 'DRIVER	User interface	Ctrl var.	5-259
DSMEMORY [*offset*, *value*, *flag*]	Hardware	Function	5-268
EIGHTH [*list*]	Selector	Function	5-7
ENDP [*object*]	Recognizer	Function	5-60
EQ [obj_1, obj_2]	Comparator	Function	5-61
EQL [obj_1, obj_2]	Comparator	Function	5-62
EQUAL [obj_1, obj_2, *test*]	Comparator	Function	5-63
EVAL [*object*]	Evaluator	Function	5-217
EVAL-FUNCTION-P [*symbol*]	Evaluator	Function	5-214
EVENP [*object*]	Numerical	Function	5-116
EVERY [*test*, $list_1$, ..., $list_n$]	Evaluator	Function	5-230
EXACT-MODE: NIL	Numerical	Ctrl var.	5-111
EXECUTE [*program*, *command*]	Control	Function	5-249

EXP [*n*]	Numerical	Function	5-134
EXPT [*n*, *m*]	Numerical	Function	5-135
FBOUNDP [*symbol*]	Evaluator	Function	5-214
FIFTH [*list*]	Selector	Function	5-7
FILE-DATE [*file*]	File	Function	5-207
FILE-LENGTH [*file*]	File	Function	5-207
FILE-LINE-LENGTH [*n*, *file*]	File	Function	5-205
FILE-READ-POSITION [*n*, *file*]	File	Function	5-196
FILE-WRITE-POSITION [*n*, *file*]	File	Function	5-204
FILL [*list*, *object*]	Modifier	Function	5-40
FIND [*object*, *list*, *test*]	Searcher	Function	5-14
FIND-IF [*test*, *list*]	Searcher	Function	5-14
FINDSTRING [$atom_1$, $atom_2$, *n*]	String	Function	5-107
FIRST [*list*]	Selector	Function	5-7
FIRSTN [*n*, *list*]	Constructor	Function	5-23
FLAG [*symbol*, *attribute*]	Flag	Function	5-80
FLAGP [*symbol*, *attribute*]	Flag	Function	5-80
FLOOR [*n*, *m*]	Numerical	Function	5-144
FMAKUNBOUND [*symbol*]	Definition	Function	5-86
FOREGROUND-COLOR [*n*]	Screen	Function	5-281
FORWARD [*distance*]	Graphics	Function	6-21
FOURTH [*list*]	Selector	Function	5-7
FREE-LIST: NIL	Memory	Variable	5-253
FRESH-LINE [*sink*]	Output	Function	5-181
FUNCALL [*function*, arg_1, ... arg_n]	Evaluator	Function	5-221
FUNCTIONP [*object*]	Evaluator	Function	5-215
GCD [n_1, n_2, ..., n_m]	Numerical	Function	5-131
GCDTRAP [*object1*, *object2*]	Numerical	Function	5-158
GCHOOK: NIL	Memory	Ctrl var.	5-253
GCSTAT: NIL	Memory	Ctrl var.	5-251
GENSYM [atom]	String	Function	5-109
GENSYM-COUNT: 0	String	Ctrl var.	5-109
GENSYM-PREFIX: G	String	Ctrl var.	5-109
GET [*symbol*, *key*]	Property	Function	5-78

GET-BREAK-CHARS []	Input	Function	5-16
GET-DECODED-TIME []	Miscellaneous	Function	5-[illegible]
GET-MACRO-CHAR [*char*, *flag*]	Input	Function	5-171
GETD [*symbol*, *flag*]	Definition	Function	5-8[illegible]
GETSET [*parameter*]	Miscellaneous	Function	5-[illegible]
:GETTABLE-INSTANCE-VARIABLES	Flavors	Option	6-1[illegible]
HIDE-MOUSE []	Mouse	Function	6-24
HIGH-INTENSITY: NIL	Screen	Ctrl var.	5-287
IDENTITY [*object*]	Control	Function	5-237
IF [*testform*, *thenform*, *elseform*]	Control	Spcl form	5-238
IGNORE-CASE: NIL	Input	Ctrl var.	5-176
INCF [*place*, *n*]	Numerical	Macro	5-130
:INCLUDE [*structure*]	Structure	Option	6-8
:INCLUDED-FLAVORS [$flav_1$, ...]	Flavors	Option	6-12
INCQ [*symbol*, *n*]	Numerical	Spcl form	5-129
:INITABLE-INSTANCE-VARIABLES	Flavors	Option	6-12
:INITIAL-CONTENTS *expression*	Array	Option	6-7
:INITIAL-ELEMENT *expression*	Array	Option	6-7
INPUT-ECHO: NIL	Input	Ctrl var.	5-175
INPUT-FILE [*file*]	File	Function	5-193
INPUT-FILE: NIL	File	Ctrl var.	5-195
INPUT-FILE-P [*file*]	File	Function	5-194
INPUT-FILES []	File	Function	5-194
INSERT-LINES [*n*]	Screen	Function	5-286
INSERT-MODE: T	User interface	Ctrl var.	5-267
INSERT-NTH [*item*, *list*, *n*]	Modifier	Function	5-45
INTEGER-LENGTH [*n*]	Numerical	Function	5-153
INTEGERP [*object*]	Recognizer	Function	5-56
INTERRUPT [*n*]	Hardware	Function	5-272
INTERRUPT-HOOK	Evaluator	Ctrl var.	5-227
INTERSECTION [$list_1$, $list_2$, *test*]	Constructor	Function	5-34
ISQRT [*n*]	Numerical	Function	5-137

LAMBDA	Definition	Keyword	5-83
LAST [*list*]	Selector	Function	5-8
LCM [n_1, ..., n_m]	Numerical	Function	5-133
LCMTRAP [$object_1$, $object_2$]	Numerical	Function	5-158
LCONC [*dotted-pair*, *list*]	Modifier	Function	5-51
LDIFF [*list*, *tail*]	Constructor	Function	5-25
LEFT [*angle*]	Graphics	Function	6-21
LEFT-BUTTON	Mouse	Variable	6-26
LENGTH [*object*]	Selector	Function	5-10
LET [*letlist*, $form_1$, ... $form_n$]	Control	Macro	5-241
LET* [*letlist*, $form_1$, ... $form_n$]	Control	Macro	5-241
LET-CLOSED [*letlist*, *function*]	Closures	Macro	6-31
LINE [x_1, y_1, x_2, y_2]	Graphics	Function	6-19
LINE-COLUMN	User interface	Variable	5-264
LINE-EDIT [*text*, *point*, *col*, *width*]	User interface	Function	5-264
LINE-POINT	User interface	Variable	5-264
LINE-TERMINATOR	User interface	Variable	5-264
LIST [obj_1, ..., obj_n]	Constructor	Function	5-19
LIST* [obj_1, ..., obj_n]	Constructor	Function	5-19
LIST-LENGTH [*object*]	Selector	Function	5-10
LISTEN [*source*]	Input	Function	5-165
LISTP [*object*]	Recognizer	Function	5-59
LN [*n*]	Numerical	Function	5-136
LOAD [*file*, *display*]	Environment	Function	5-211
LOAD-VERBOSE: T	Environment	Ctrl var.	5-211
LOCATION [*object*]	Hardware	Function	5-271
LOG [*n*, *base*]	Numerical	Function	5-136
LOGAND [n_1, ..., n_m]	Numerical	Function	5-151
LOGBITP [*index*, *n*]	Numerical	Function	5-150
LOGEQV [n_1, ..., n_m]	Numerical	Function	5-151
LOGIOR [n_1, ..., n_m]	Numerical	Function	5-151
LOGNOT [*n*]	Numerical	Function	5-150
LOGTEST [*n*, *m*]	Numerical	Function	5-150
LOGXOR [n_1, ..., n_m]	Numerical	Function	5-151
LOOP [$form_1$, ..., $form_n$]	Control	Spcl form	5-235
LOWER-CASE-P [*symbol*]	Character	Function	5-90

MACRO	Definition	Keyword	5-1
MACRO-FUNCTION [*symbol*]	Evaluator	Function	5-2
MACRO-FUNCTION-P [*symbol*]	Evaluator	Function	5-21
MACROEXPAND [*form*]	Evaluator	Function	5-2
MACROEXPAND: T	Evaluator	Ctrl var.	5-2.
MACROEXPAND-1 [*form*]	Evaluator	Function	5-2
MAKE-ARRAY [*dims*, *opt*$_1$, ...]	Array	Function	6
MAKE-INSTANCE [*flavor*, *vars*]	Flavors	Macro	6-1
MAKE-LIST [*n*, *object*, *list*]	Constructor	Function	5-2
MAKE-RANDOM-STATE [*state*]	Numerical	Function	5-1
MAKE-WINDOW [*row*, *col*, *rows*, *cols*]	Screen	Function	5-2
MAPC [*function*, *list*$_1$, ...]	Evaluator	Function	5-2
MAPCAN [*function*, *list*$_1$, ...]	Evaluator	Function	5-2
MAPCAR [*function*, *list*$_1$, ...]	Evaluator	Function	5-2
MAPCON [*function*, *list*$_1$, ...]	Evaluator	Function	5-2
MAPL [*function*, *list*$_1$, ...]	Evaluator	Function	5-2
MAPLIST [*function*, *list*$_1$, ...]	Evaluator	Function	5-2
MAX [n_1, ..., n_m]	Numerical	Function	5-1
MEMBER [*object*, *list*, *test*]	Searcher	Function	5-1
MEMBER-IF [*test*, *list*]	Searcher	Function	5-1
MEMORY [*address*, *value*, *flag*]	Hardware	Function	5-2
MERGE [*list*$_1$, *list*$_2$, *test*]	Modifier	Function	5-
:METHOD-COMBINATION	Flavors	Option	6-1
MIN [n_1, ..., n_m]	Numerical	Function	5-1
MINUSP [*object*]	Numerical	Function	5-1
MISMATCH [*list*$_1$, *list*$_2$, *test*]	Searcher	Function	5-
MOD [*n*, *m*]	Numerical	Function	5-1
MOUSE-BOX [x_1, y_1, x_2, y_2]	Mouse	Function	6-
MOUSE-POSITION [*x*, *y*]	Mouse	Function	6-
MOUSE-RESET []	Mouse	Function	6-
MOUSE-STATUS []	Mouse	Function	6-
MOUSE-X	Mouse	Function	6-
MOUSE-Y	Mouse	Function	6-
MOVD [*symbol*$_1$, *symbol*$_2$]	Definition	Function	5-

:NAMED	Structure	Option	6-9
NBUTLAST [*list*, *n*]	Modifier	Function	5-42
NCONC [$list_1$, ..., $list_n$]	Modifier	Function	5-41
NEQ [obj_1, obj_2]	Comparator	Function	5-61
NEQL [obj_1, obj_2]	Comparator	Function	5-62
NEW-CODE-SPACE []	Memory	Function	5-255
NIL	Evaluator	Constant	5-197
NINTH [*list*]	Selector	Function	5-7
NLAMBDA	Definition	Keyword	5-83
NO-EVAL-FUNCTION-P [*symbol*]	Evaluator	Function	5-214
:NO-VANILLA-FLAVOR	Flavors	Option	6-13
NOT [*object*]	Logical	Function	5-66
NOTANY [*test*, $list_1$, ...]	Evaluator	Function	5-229
NOTEVERY [*test*, $list_1$, ...]	Evaluator	Function	5-230
NRECONC [*list*, *object*]	Modifier	Function	5-43
NREVERSE [*list*, *object*]	Modifier	Function	5-43
NSUBLIS [*alist*, *object*, *test*]	Modifier	Function	5-49
NSUBST [*new*, *old*, *object*, *test*]	Modifier	Function	5-48
NSUBST-IF [*new*, *test*, *object*]	Modifier	Function	5-48
NSUBSTITUTE [*new*, *old*, *list*, *test*]	Modifier	Function	5-47
NSUBSTITUTE-IF [*new*, *test*, *list*]	Modifier	Function	5-47
NTH [*n*, *list*]	Selector	Function	5-9
NTHCDR [*n*, *list*]	Selector	Function	5-9
NULL [*object*]	Recognizer	Function	5-58
NUMBERP [*object*]	Recognizer	Function	5-56
NUMERATOR [*n*]	Numerical	Function	5-143
NUMERIC-CHAR-P [*symbol*]	Character	Function	5-91
NUMTRAP [*object*]	Numerical	Function	5-158
OBLIST []	Constructor	Function	5-37
ODDP [*object*]	Numerical	Function	5-116
OPEN-INPUT-FILE [*file*]	File	Function	5-191
OPEN-OUTPUT-FILE [*file*, *overwrite*]	File	Function	5-198
OR [$form_1$, ..., $form_n$]	Logical	Spcl form	5-68
ORDERP [$object_1$, $object_2$]	Comparator	Function	5-65
OUTPUT-ECHO: NIL	Output	Ctrl var.	5-184

OUTPUT-FILE [*file*]	File	Function	5-101
OUTPUT-FILE: NIL	File	Ctrl var	5-70
OUTPUT-FILE-P [*file*]	File	Function	5-70
OUTPUT-FILES []	File	Function	5-70
PACK [*list*]	String	Function	5-9
PACK* [$atom_1$, ..., $atom_n$]	String	Function	5-9
PAIRLIS [*keys*, *objects*, *alist*]	Constructor	Function	5-3
PALETTE-COLOR [*n*]	Screen	Function	5-28
PARSE-INTEGER [*string*, *radix*]	Input	Function	5-16
PEEK-CHAR [*type*, *source*]	Input	Function	5-16
PENDOWN []	Graphics	Function	6-2
PENUP []	Graphics	Function	6-2
PI []	Numerical	Function	5-14
PLOT-CIRCLE [*x*, *y*, *r*, *color*]	Screen	Function	5-28
PLOT-DOT [*x*, *y*, *color*]	Screen	Function	5-28
PLOT-LINE [x_1, y_1, x_2, y_2, *color*]	Screen	Function	5-28
PLUSP [*object*]	Numerical	Function	5-11
POP [*symbol*]	Assignment	Spcl form	5-7
PORTIO [*port*, *value*, *flag*]	Hardware	Function	5-27
POSITION [*object*, *list*, *test*]	Searcher	Function	5-1
POSITION-IF [*test*, *list*]	Searcher	Function	5-1
PRECISION [*n*]	Numerical	Function	5-11
:PREDICATE [*name*]	Structure	Option	6-
PRIN1 [*object*, *sink*]	Output	Function	5-17
PRINC [*object*, *sink*]	Output	Function	5-17
PRINT [*object*, *sink*]	Output	Function	5-17
PRINT-BASE: 10	Output	Ctrl var.	5-18
PRINT-DOWNCASE: NIL	Output	Ctrl var.	5-18
PRINT-ESCAPE: T	Output	Ctrl var.	5-18
PRINT-LENGTH [*atom*]	String	Function	5-10
PRINT-POINT: 7	Output	Ctrl var.	5-18
PRINTER-ECHO: NIL	Output	Ctrl var.	5-18
PROBE-FILE [*file*]	File	Function	5-20
PROG1 [$form_1$, ... $form_n$]	Control	Spcl form	5-22
PROG2 [$form_1$, ... $form_n$]	Control	Macro	5-23

PROGN [$form_1$, ... $form_n$]	Control	Spcl form	5-234
PSETQ [$symbol_1$, $form_1$, ...]	Assignment	Spcl form	5-72
PUSH [*form*, *symbol*]	Assignment	Spcl form	5-75
PUSHNEW [*form*, *symbol*, *test*]	Assignment	Macro	5-75
PUT [*symbol*, *key*, *object*]	Property	Function	5-77
PUTD [*symbol*, *definition*]	Definition	Function	5-84
QUOTE [*object*]	Control	Spcl form	5-233
RANDOM [*n*, *state*]	Numerical	Function	5-154
RANDOM-STATE	Numerical	Ctrl var.	5-155
RANDOM-STATE-P [*object*]	Numerical	Function	5-156
RASSOC [*key*, *alist*, *test*]	Searcher	Function	5-17
RASSOC-IF [*test*, *alist*]	Searcher	Function	5-17
RATIONAL	Numerical	Keyword	5-112
RATIONALP [*object*]	Recognizer	Function	5-56
READ [*source*]	Input	Function	5-160
READ-ATOM [*source*]	Input	Function	5-161
READ-BASE: 10	Input	Ctrl var.	5-175
READ-BYTE [*source*]	Input	Function	5-163
READ-CHAR [*source*]	Input	Function	5-163
READ-DOT [*x*, *y*]	Screen	Function	5-286
READ-FROM-STRING [*string*]	Input	Function	5-167
READ-LINE [*source*]	Input	Function	5-163
READ-STRINGS	Input	Variable	5-167
READ-UPCASE: T	Input	Ctrl var.	5-175
RECLAIM []	Memory	Function	5-250
REDUCE [*function*, *list*, *init*]	Evaluator	Function	5-231
REGISTER [*n*, *m*]	Hardware	Function	5-272
REM [*n*, *m*]	Numerical	Function	5-148
REMD [*symbol*]	Definition	Function	5-86
REMFLAG [*symbol*, *attribute*]	Flag	Function	5-81
REMOVE [*item*, *list*, *test*]	Constructor	Function	5-28
REMOVE-DUPLICATES [*list*, *test*]	Constructor	Function	5-29
REMOVE-IF [*test*, *list*]	Constructor	Function	5-28
REMPROP [*symbol*, *key*]	Property	Function	5-79

RENAME-FILE [*oldname*, *newname*]	File	Function	5-20
REPLACE [$list_1$, $list_2$]	Modifier	Function	5-4
REPLACE-NTH [*item*, *list*, *n*]	Modifier	Function	5-4
:REQUIRED-FLAVORS [$flav_1$, ...]	Flavors	Option	6-1
REST [*list*]	Selector	Function	5-
RESTART []	Control	Function	5-24
RETURN [*form*]	Control	Spcl form	5-24
REVAPPEND [*list*, *object*]	Constructor	Function	5-2
REVERSE [*list*, *object*]	Constructor	Function	5-2
RIGHT [*angle*]	Graphics	Function	6-21
RIGHT-BUTTON	Mouse	Variable	6-2
ROUND [*n*, *m*]	Numerical	Function	5-14
ROW []	Screen	Function	5-27
RPLACA [obj_1, obj_2]	Modifier	Function	5-3
RPLACD [obj_1, obj_2]	Modifier	Function	5-3
SAVE [*file*]	Environment	Function	5-20
SECOND [*list*]	Selector	Function	5-
SELF	Flavors	Variable	6-1
SEND [*obj*, *op*, arg_1, ...]	Flavors	Macro	6-1
SET [*symbol*, *object*]	Assignment	Function	5-7
SET-BREAK-CHARS [*list*, *flag*]	Input	Function	5-16
SET-CURSOR [*row*, *column*]	Screen	Function	5-27
SET-DIFFERENCE [$list_1$, $list_2$, *test*]	Constructor	Function	5-3
SET-EXCLUSIVE-OR [$list_1$, $list_2$, *test*]	Constructor	Function	5-3
SET-IN-CLOSURE [*closure*, *sym*, *obj*]	Closures	Function	6-3
SET-MACRO-CHAR [*char*, *defn*, *flg*]	Input	Function	5-17
SETCOLOR [*angle*]	Graphics	Function	6-2
SETF [$place_1$, $form_1$, ...]	Assignment	Macro	5-7
SETHEADING [*angle*]	Graphics	Function	6-2
SETPOS [*x*, *y*]	Graphics	Function	6-2
SETQ [$symbol_1$, $form_1$, ...]	Assignment	Spcl form	5-7
:SETTABLE-INSTANCE-VARIABLES	Flavors	Option	6-1
SEVENTH [*list*]	Selector	Function	5-
SHIFT [*n*, *count*]	Numerical	Function	5-15
SHOW-MOUSE []	Mouse	Function	6-2

SIGNUM [*n*]	Numerical	Function	5-142
SIN [*n*]	Numerical	Function	5-138
SIN-DEG [*angle*]	Graphics	Function	6-18
SIXTH [*list*]	Selector	Function	5-7
SNAPSHOT [*address*, *atom*]	Hardware	Function	5-275
SOME [*test*, $list_1$, ... $list_n$]	Evaluator	Function	5-229
SORT [*list*, *test*]	Modifier	Function	5-54
SPACES [*n*, *sink*]	Output	Function	5-182
SPECIAL	Definition	Keyword	5-83
SPECIAL-FORM-P [*symbol*]	Evaluator	Function	5-214
SPLIT [*list*]	Modifier	Function	5-52
SQRT [*n*]	Numerical	Function	5-137
STABLE-SORT [*list*, *test*]	Modifier	Function	5-54
STACK-LIST []	Constructor	Function	5-37
STRING-CAPITALIZE [*atom*]	String	Function	5-106
STRING-DOWNCASE [*atom*]	String	Function	5-106
STRING-EQUAL [$atom_1$, $atom_2$]	String	Macro	5-102
STRING-GREATERP [$atom_1$, $atom_2$]	String	Macro	5-104
STRING-INDEX	Input	Variable	5-167,8
STRING-LEFT-TRIM [*chars*, *atom*]	String	Function	5-105
STRING-LESSP [$atom_1$, $atom_2$]	String	Macro	5-104
STRING-NOT-EQUAL [$atom_1$, $atom_2$]	String	Macro	5-104
STRING-NOT-GREATERP [atm_1, atm_2]	String	Macro	5-104
STRING-NOT-LESSP [$atom_1$, $atom_2$]	String	Macro	5-104
STRING-RIGHT-TRIM [*chars*, *atom*]	String	Function	5-105
STRING-TRIM [*chars*, *atom*]	String	Function	5-105
STRING-UPCASE [*atom*]	String	Function	5-106
STRING= [$atom_1$, $atom_2$, *flag*]	String	Function	5-102
STRING< [$atom_1$, $atom_2$, *flag*]	String	Function	5-103
STRING> [$atom_1$, $atom_2$, *flag*]	String	Function	5-103
STRING<= [$atom_1$, $atom_2$, *flag*]	String	Function	5-103
STRING>= [$atom_1$, $atom_2$, *flag*]	String	Function	5-103
STRING/= [$atom_1$, $atom_2$, *flag*]	String	Function	5-103
STRINGP [*object*]	Recognizer	Function	5-55
SUB1 [*n*]	Numerical	Function	5-128
SUBLIS [*alist*, *object*, *test*]	Constructor	Function	5-32

SUBLIST [*list*, *n*, *m*]	Constructor	Function	5-71
SUBSETP [$list_1$, $list_2$, *test*]	Comparator	Function	5-61
SUBST [*new*, *old*, *object*, *test*]	Constructor	Function	5-31
SUBST-IF [*new*, *test*, *object*]	Constructor	Function	5-31
SUBSTITUTE [*new*, *old*, *list*, *test*]	Constructor	Function	5-30
SUBSTITUTE-IF [*new*, *test*, *list*]	Constructor	Function	5-30
SUBSTRING [*atom*, *n*, *m*]	String	Function	5-100
SYMBOL-FUNCTION [*symbol*]	Evaluator	Function	5-216
SYMBOL-PLIST [*symbol*]	Evaluator	Function	5-216
SYMBOL-VALUE [*symbol*]	Evaluator	Function	5-216
SYMBOLP [*object*]	Recognizer	Function	5-55
SYMEVAL-IN-CLOSURE [*closures*, *sym*]	Closures	Function	6-33
SYSTEM [*n*]	Control	Function	5-248
TAB-EXPAND: NIL	Input	Ctrl var.	5-176
TAILP [$list_1$, $list_2$]	Comparator	Function	5-64
TAN [*n*]	Numerical	Function	5-118
TCONC [*dotted-pair*, *object*]	Modifier	Function	5-50
TENTH [*list*]	Selector	Function	5-7
TERPRI [*n*, *sink*]	Output	Function	5-181
THIRD [*list*]	Selector	Function	5-7
THROW [*tag*, *form*]	Control	Spcl form	5-217
THROW-TAG	Control	Variable	5-217
TIME [*flag*]	Miscellaneous	Function	5-239
TONE [*frequency*, *duration*]	Miscellaneous	Function	5-238
TREE-EQUAL [$object_1$, $object_2$, *test*]	Comparator	Function	5-63
TRUNCATE [*n*, *m*]	Numerical	Function	5-116
TURTLE []	Graphics	Function	6-22
TURTLE-WINDOW []	Graphics	Function	6-22
:TYPE [LIST]	Structure	Option	6-9
UNDEFINED [*symbol*, $form_1$, ...]	Evaluator	Function	5-222
UNDERFLOW [*n*]	Numerical	Function	5-113
UNION [$list_1$, $list_2$, *test*]	Constructor	Function	5-34
UNLESS [*testform*, $form_1$, ..., $form_n$]	Control	Macro	5-238
UNPACK [*atom*]	String	Function	5-98

UNREAD-CHAR [*source*]	Input	Function	5-164
UNWIND-PROTECT [$form_1$, $form_2$, ...]	Control	Spcl form	5-246
UPPER-CASE-P [*symbol*]	Character	Function	5-90
VECTOR [$object_1$, $object_2$, ...]	Array	Function	6-2
VIDEO-MODE [*n*]	Screen	Function	5-283
WHEN [*testform*, $form_1$, ..., $form_n$]	Control	Macro	5-238
WRITE [*object*, *sink*]	Output	Function	5-178
WRITE-BYTE [*n*, *m*, *sink*]	Output	Function	5-183
WRITE-LINE [*symbol*, *sink*]	Output	Function	5-180
WRITE-STRING [*symbol*, *sink*]	Output	Function	5-180
Y-OR-N-P [*message*]	User interface	Function	5-263
YES-OR-NO-P [*message*]	User interface	Function	5-263
ZEROP [*object*]	Numerical	Function	5-114
+ [n_1, ..., n_m]	Numerical	Function	5-124
- [n_1, ..., n_m]	Numerical	Function	5-125
* [n_1, ..., n_m]	Numerical	Function	5-126
/ [n_1, ..., n_m]	Numerical	Function	5-127
+TRAP [$object_1$, $object_2$]	Numerical	Function	5-157
-TRAP [$object_1$, $object_2$]	Numerical	Function	5-157
*TRAP [$object_1$, $object_2$]	Numerical	Function	5-157
/TRAP [$object_1$, $object_2$]	Numerical	Function	5-157
= [n_1, ..., n_m]	Numerical	Function	5-117
/= [n_1, ..., n_m]	Numerical	Function	5-118
< [n_1, ..., n_m]	Numerical	Function	5-119
> [n_1, ..., n_m]	Numerical	Function	5-120
<= [n_1, ..., n_m]	Numerical	Function	5-121
>= [n_1, ..., n_m]	Numerical	Function	5-122

\ back slash	Input	Esc char.	5-16
| vertical bar	Input	Esc char.	5-16
(left parenthesis	Input	Macro char.	5-17
) right parenthesis	Input	Macro char.	5-17
] right square bracket	Input	Macro char.	5-17
, comma	Input	Macro char.	5-17
' single quote	Input	Macro char.	5-17
" double quote	Input	Macro char.	5-17
; semicolon	Input	Macro char.	5-17
` backquote	Input	Macro char.	5-17
+	User interface	Variable	5-25
++	User interface	Variable	5-25
+++	User interface	Variable	5-25
*	User interface	Variable	5-25
**	User interface	Variable	5-25
***	User interface	Variable	5-25
-	User interface	Variable	5-25

Soft Warehouse License Agreement and Warranty

This agreement is between you, the end user, and Soft Warehouse, Inc. BY OPENING THE SEALED DISK ENVELOPE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS AGREEMENT, WHICH INCLUDES THE SOFTWARE LICENSE, SOFTWARE DISCLAIMER OF WARRANTY, AND THE LIMITED WARRANTY ON THE DISKETTES AND WRITTEN MATERIALS.

This agreement constitutes the entire agreement between you and Soft Warehouse, Inc. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, DO NOT OPEN THE SEALED DISK ENVELOPE. Promptly return the unopened envelope and the rest of the materials that make up the software package (including written documentation) to the place of purchase for a refund.

Grant of License: Part of the price you paid for this product is the License Fee. Soft Warehouse, Inc. grants you, the LICENSEE, a nonexclusive right to use this software program and associated written documentation (together called the SOFTWARE), according to the following terms and conditions:

Term: This Agreement is effective upon receipt of the SOFTWARE by you and shall remain in force until terminated by you or by Soft Warehouse, Inc. This License will terminate automatically without notice if you fail to comply with any provision of this License. Upon termination you shall destroy all copies of the SOFTWARE.

Title and Ownership: The SOFTWARE is, and shall at all times remain, the property of and copyrighted by Soft Warehouse, Inc. As LICENSEE, you own the physical media on which the SOFTWARE is recorded for transfer to you. Soft Warehouse, Inc. retains title and ownership of any and all copies of the SOFTWARE recorded on any physical media.

Copy Restrictions: This SOFTWARE, including the accompanying written documentation, is copyrighted. Unauthorized copying of the SOFTWARE, including SOFTWARE that has been modified, merged, or included with other software, is expressly forbidden. You may be held legally responsible for any copyright infringement that is caused or encouraged by your failure to abide by the terms of this License.

Archival Copies: You may make archival copies of those portions of the SOFTWARE that are provided in machine readable form, provided that such copies are for archival or backup purposes only and that they will be destroyed when and if this Agreement is terminated. You must reproduce the Soft Warehouse, Inc. copyright notice on all copies.

IMPORTANT! Terms of Use Continued on Reverse

Protection, Security, and Temporary Use: The SOFTWARE is licensed for the exclusive use of you, the LICENSEE, who shall hold the SOFTWARE in confidence. You may not distribute copies of the SOFTWARE to others except as provided herein. The SOFTWARE may be run on at most five (5) computers or five (5) computer terminals in a network, all of which must be under the direct control of LICENSEE and reside in a single building or facility. At this single site, LICENSEE is authorized to allow temporary use of the SOFTWARE by LICENSEE's family, employees, consultants, or students, provided LICENSEE: (a) takes all reasonable steps to ensure that the SOFTWARE is not disclosed, duplicated, or used in contravention of the provisions of this Agreement by such persons; (b) takes all reasonable steps to recover or destroy copies of the SOFTWARE made for such temporary use as soon as they are no longer necessary; (c) immediately notifies Soft Warehouse, Inc. in writing of each failure to recover or destroy any copy.

Transfer Restrictions: This SOFTWARE is licensed to you, the LICENSEE, and may not be transferred to anyone without prior written consent of Soft Warehouse, Inc. Any authorized transferee of the SOFTWARE shall be bound by the terms and conditions of this Agreement. In no event may you transfer, assign, rent, lease, sell, or otherwise dispose of the SOFTWARE on a temporary or permanent basis except as expressly provided herein.

Limited Warranty: Soft Warehouse, Inc. warrants that all materials furnished constitute an accurate copy of the SOFTWARE. Soft Warehouse, Inc. will replace defective machine readable media or written documentation, provided: (a) the defect is found within thirty (30) days of receipt of the SOFTWARE; (b) LICENSEE has not subjected the materials to abuse; (c) the media have not been used in defective or incompatible equipment; (d) the defective materials are promptly returned to Soft Warehouse, Inc.

Limitation of Liability: SOFT WAREHOUSE, INC., MAKES NO OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE SOFTWARE'S MERCHANTABILITY, ITS FITNESS FOR A PARTICULAR PURPOSE, AND ITS PERFORMANCE OR ACCURACY. Soft Warehouse, Inc. shall not be liable for any expense, claim, liability, loss or damage (including any incidental or consequential damage) either direct or indirect, by LICENSEE or by third parties, from the use of the SOFTWARE. In any event Soft Warehouse, Inc.'s liability to LICENSEE on any ground, including but not limited to negligence, shall not exceed the fee paid to Soft Warehouse, Inc. by LICENSEE.

Governing Law: This Agreement shall be interpreted and governed by the laws of the State of Hawaii. If any provision of this Agreement is invalidated by applicable statute or rule of law, the remainder of this Agreement shall remain in binding effect.

Entire Agreement: This Agreement constitutes the entire agreement between the parties with respect to the SOFTWARE and supersedes any prior agreement.